Locally least-cost error repair in LR parsers

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Doctor of Philosophy
in the
University of Canterbury
by
Carl Cerecke

Examining Committee

Dr. Tim Bell	University of Canterbury
Dr. Robert Biddle	Victoria University, Wellington
Prof. Allen Tucker	Bowdoin College

University of Canterbury
2003

To my wife Carolyn, and my children Esther, Abigail, and James.

Abstract

This thesis presents some methods for improving the efficiency and effectiveness of locally least-cost error repair algorithms for an LR-based parser. Three different algorithms for reducing the search space are described and compared using a collection of 59,643 incorrect Java programs collected from novice programmers. Two of the algorithms prove particularly effective at reducing the search space. Also presented is a more efficient priority queue implementation for storing transformations of the input string. The effect on repairs of different grammars describing the same language is investigated, and a comparison of different methods of assigning costs to edit operations is performed.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Thank you to my wife, Carolyn, for her unwavering support of my thesis, despite often difficult circumstances. Thank you to my children Esther (6yrs), Abigail (4yrs), and James (2yrs) for enriching my life, and helping to keep my priorities straight. Thank you to Bruce, my first supervisor, for introducing me to the topic and for guiding me at the start of the thesis. Thank you to Paddy for taking over when Bruce left, and helping to get me focused on finishing my thesis. Thank you to Tim, for taking over when Paddy left and for reading drafts of this thesis, despite your numerous other responsibilities. Thank you to the friendly staff at the Department of Computer Science for their encouragement, and the opportunities to work part time in various roles to support my family. Thank you to Thomas and Kathrin at Da Vinci Communications Ltd. for allowing me time at work to complete the thesis. Finally, thank you to God, whose crazy idea this was in the first place.

Chapter I

Introduction

In writing a computer program the programmer must obey the formal rules of the programming language in order for the compiler to be able to compile the program. Because humans are often not as precise as a computer, the formal rules of a programming language are not always completely followed when writing a program in that language, which results in compile-time errors. Because such errors in programs are inevitable, and can represent a misunderstanding of the formal rules of the language, good handling of these errors should be a design goal of all compilers.

Errors in a program can occur at several different levels: lexical (e.g. misspellings such as `whlie` instead of `while`), syntax (e.g. an incorrectly specified `for` loop), semantics (e.g. use of an uninitialised variable), and logical (e.g. an infinite loop). This thesis is concerned primarily with syntax errors. Parsing correct sentences of a formally specified language is well understood, and many tools are available for automatically generating a parser from such a language specification. Handling incorrect sentences while parsing is, however, still seen as somewhat of a "black art".

The problem, then, is to have a good method for handling incorrect sentences. Desirable properties of error handlers include [42]:

- Applicable to a wide range of languages. Ideally, the method of handling incorrect sentences of a language should be independent of the language. In practice, fine tuning for a particular language is often required, and should be reasonably simple.

- Able to be automatically constructed from the formal language speci-

fication.

- Should not significantly slow down the parsing of correct sentences.
- Should not use resources excessively while repairing.
- Able to restart parsing after an error, to catch other errors. Restarting a parse should not lead to spurious errors.
- A minimal amount of input should be skipped.

The goal of this thesis is to describe an error handling technique with all of the above properties.

1.1 Preliminaries

First, the question "What is an error?" must be answered. For the purposes of this thesis, an error is a point in a parse where the current state of the parser can not parse the next input symbol. Thus we have an *error stack*, the stack where the parser has no legal action, and an *error symbol*, the token for which there is no action from the state (the *error state*) on top of the stack.

Second, the question of "How should an error be handled?" is important. One possible solution is simply to discontinue parsing altogether, although this solution is likely to be unpopular with those who have more than one error in their input.

For example, a project the author was recently working on using the Java language involved two classes implementing an interface. The interface and one of the implementing classes were developed in close association, while the other implementing class was only updated periodically to reflect changes in the interface. Often a dozen or more changes had taken place in the interface, which required attention in the second implementing class, yet the Java compiler would only report that the class did not correctly implement the interface and show only the first offending construct. It was particularly frustrating, not only because the compiler had to be re-run for each error,

but because the errors were often relatively trivial and it would not have been difficult for the compiler to continue parsing and give feedback for each error because helping the programmer with his or her obvious mistakes is what a compiler *should* do.

A better solution to discontinuing parsing is to do some sort of "error recovery", that is, attempt to restart the parsing somehow, in order to process the remaining input. Restarting the parsing must be done with minimal disruption to future input. For example, deleting the remaining input and inserting a minimal suffix for a correct program would certainly allow parsing to continue, but is no better than simply discontinuing parsing.

1.2 Convention and Notation

The algorithms presented in this thesis have been implemented within *bison* [23], a widely used LALR(1) parser generator. Where an example uses the LALR parsing automaton produced by *bison*, these automata are often represented in the form of a graph automatically produced directly from the debugging output produced by *bison* as a by-product of creating a parser. An example is shown in Figure 2.1 on page 9.

Rectangular boxes indicate states in the parser. The state number is shown within each box, along with the core items of that state. The accepting state is displayed in the shape of a triangle. Ovals represent reductions; the relevant reduction is displayed within the oval. Shift actions and goto actions are displayed as edges between states. The end-of-input symbol is denoted as '$'.

1.2.1 Terms used

The literature uses a variety of error handling terms interchangeably, resulting in some confusion as to the meaning of these terms. The terminology in this thesis is based on that of Grune and Jacobs [41] with influence from Dain [26].

Error handling is used to describe in general what a parser does on encountering a syntax error.

Error recovery is an error handling scheme that attempts to recover from the syntax error and continue parsing.

Error repair is a method of error recovery where the incorrect input string is transformed into a correct string to allow parsing to continue.

Error correction is altering the input so that it becomes what the user intended. Clearly this is impossible for an automatic error repair method in general[1]; error correction is a task for the user. A number of authors use the term *correction* for what has been described here as error repair.

Validating a possible repair involves successfully parsing some given number of input symbols following the repair. The number of input symbols that must be successfully parsed following a repair is referred to as the *validation length.*

1.3 Thesis outline

The reader is assumed to be familiar with context free grammars, and LR-parsing theory to the level described in a compiler textbook such as the 'Dragon book' [3]. Chapter 2 describes previous work in error handling, with a particular emphasis on local error recovery. Chapter 3 describes in more detail the base algorithm from McKenzie et al. [59] to which the pruning algorithms developed in chapters 4, 5, and 6 are applied. Chapter 4 describes the first of the three pruning algorithms. It allows for the insertion of non-terminals during a repair. Chapter 5 explains the conditions under which loops may be safely omitted from the search for a least-cost repair with a known validation length. Chapter 6 introduces the notion of equivalence between edges of a state in a parser. The chapter presents an algorithm to find many of these equivalent edges, and describes how the information can be used to prune out certain edges. Chapter 7 considers the interactions between the three pruning algorithms and shows that they are all independent of each other and can be used in combination without affecting the least-cost property of the repair algorithm. Chapter 8 describes experiments designed to compare the effectiveness of the three pruning algorithms, the difference

[1] Otherwise one could submit an empty file to the compiler and have it 'correct' the file by replacing it with the program the user 'intended' to write.

between the two Java grammars, and the difference in cost assignments. Chapter 9 presents the results from the experiments described in Chapter 8. Finally, Chapter 11 mentions some areas for possible future work, including a brief description of a promising new local repair algorithm, and Chapter 10 presents the conclusions of the thesis.

Chapter II

Previous work

This chapter begins with a brief overview of some LR-based parser generators, and the method each uses for error handling. Error detection is then discussed in Section 2.2, followed by an explanation of the classification of error recovery methods used in this thesis in Section 2.3. The remaining sections describe the important error recovery schemes presented in the literature.

2.1 Parsing and parser generators

LR parsing, defined by Knuth [52], is able to parse deterministic context-free languages (DCFLs) in linear time. The two main variants of LR parsing, SLR (Simple LR) [28] and LALR (lookahead LR) [27], parse a reasonably large subset of deterministic context-free languages with significantly smaller parsing tables than standard LR. Nearly all LR-based parser generators use the LALR variant.

An LALR(1) parser generator which generates parsers that halt as soon as they detect an error is described by Wetherell and Shannon [79].

The well-known LALR parser generator, *yacc*, is described by Johnson [46]. It uses an error recovery scheme based on the one presented in Aho and Johnson [1], described in Section 2.4.2. The parser generator *bison* [23] is a re-implementation of *yacc* from the Free Software Foundation. Both *bison* and *yacc* are implemented in, and generate, C code. Many *yacc*-like tools have been written for other programming languages.

ECP (Error Correcting Parser generator) was described by Mauney and Fischer [56], and uses a least-cost error correction technique that is presented in [32].

The LR(k) parser generator *Essence*, creates parsers by applying partial evaluation to a general parser [67]. It uses a similar error recovery technique to *yacc*.

The *ToolMaker* parser generator [65] is LALR(1) based and has a three-stage error recovery process. The first stage is single symbol correction: one symbol of insertion, deletion, or replacement is considered. Symbols have costs associated with them, and the least cost repair is used. By default, two symbols must be successfully parsed after the correction (a validation length of two). Stage two of the process involves generating least-cost strings and matching them against the remaining input. This stage is terminated if an attempt is made to delete a *fiducial symbol* (Section 2.6.1). Stage three of the error recovery process is a form of *panic mode* (Section 2.6.1).

2.1.1 Error recovery in Java compilers

Error repair techniques developed in this thesis are tested (Chapter 8) using Java programs. The error recovery techniques used in modern Java compilers are, therefore, of interest, and are mentioned here.

IBM have developed, and maintain, the *Jikes* compiler[1], designed for high performance, and for use in large programs. The compiler is written in C++, and uses an IBM in-house LALR(k) parser generator, *jikespg*. The parser generator has no built-in error recovery capabilities— any error recovery desired must be provided by the compiler writer. The *Jikes* compiler implements an ad-hoc, regional error recovery with validation: the state stack is successively truncated, and ever larger chunks of input are discarded until three tokens of the remaining input can be successfully parsed.

An older compiler, *guavac*[2], is written in C++, and uses the parser generator *bison* to generate its parser. Errors are handled using the error-token method provided by *bison*, described in general in Section 2.4.2.

The Kopi project[3] have developed a Java compiler, *KJC*, written in Java. It uses the recursive-descent parser generator *ANTLR*[4]. Error handling in

[1] `http://oss.software.ibm.com/developerworks/opensource/jikes/`
[2] Written by David Engberg, but no longer maintained
[3] `http://www.dms.at/kopi/index.html`
[4] `http://www.antlr.org/`

KJC uses the follow-set method (Section 2.6.2) provided as the default error handling mechanism of *ANTLR*.

The commonly used compiler developed by Sun[5] is proprietary, and no information is available on the error handling techniques it uses.

2.2 Error Detection

The term 'error detection' refers not only to a parser's ability to simply detect an error, but it also refers to that point in the input at which the error occurs, and the state of the parser when the error is detected.

Detecting the presence of errors in the input is the minimum requirement of a parser. If it did not reliably detect errors, then uncertainty would arise as to whether an input was really in the language or if the parser simply didn't notice an error. Since the entire point of a parser is to accept— and only accept— strings in a language, such an unreliable 'parser' would not be deserving of the name.

Detecting the point in a sentence at which a syntax error occurs is very desirable. Many parsing methods, including the two most common linear-time methods, LL and LR, possess the *correct prefix property*. That is, they will halt on the first token in the input that results in a prefix that cannot start a sentence of the language.

Even though a parser that has the correct prefix property will never consume an incorrect input token, some commonly used variants of LL and LR— strong LL(1), SLR(1) and LALR(1)— may perform parsing actions before finally halting at the error token [35, 39]. Strong LL(1) parsers may pop the top nonterminal from the parsing stack, if that nonterminal derives the empty production. SLR(1) and LALR(1) parsers may perform a reduction, or indeed a sequence of reductions, popping states off the parse stack, and pushing another state on. In both cases, the state where the actual error occurred is no longer likely to be on top of the stack. Additionally, the tables of LR-based parsers may be compacted in a way that many reductions are performed with no lookahead at all [46].

[5] `http://java.sun.com`

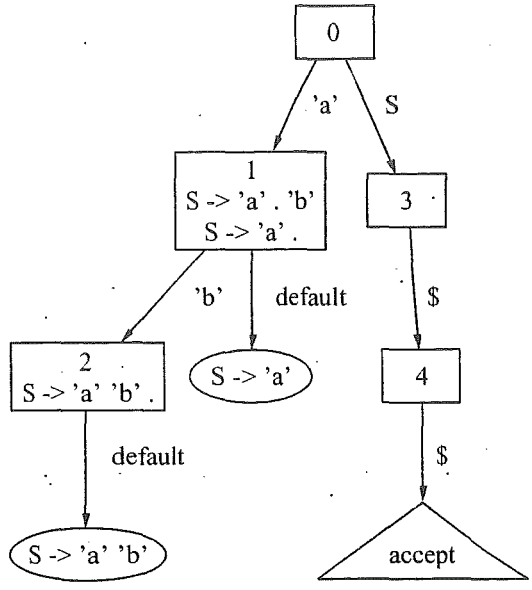

Figure 2.1: Parser illustrating the problem of reductions affecting the stack at the point where an error occurs.

Canonical LR(1), because of the extra lookahead information stored for reductions, will never perform a reduction unless the next symbol is legal. When an error occurs, the parse stack will always have the state where the error occurs on top of the stack. The term *immediate error detection property* is used for parsers which possess the correct prefix property, but also perform no actions when the next token is erroneous. In the same way, canonical LL(1) parsers have the immediate error detection property.

For an example of the problem of error detection, consider the parser shown in Figure 2.1, generated by *bison* for the grammar:

```
S → 'a' 'b'
S → 'a'
```

Bison parsers are compacted in such a way that lookahead information for reductions is completely removed for states with only one reduction. These `default` reductions are not performed during parsing because they are necessarily the correct action from that state, but because there are no other

correct actions from that state. It is impossible, therefore, for a parser generated in this way to ever halt on an error in a state with a default reduction; the default reduction will always be performed.

For the example in Figure 2.1, given the erroneous input "ac", the parser shifts to state 1 on the 'a', then, because the next symbol is not a 'b', performs the default reduction in state 1, and ends up in state 3. The opportunity to consider a repair by inserting a 'b' from state 1 is lost.

A solution to this problem, described by Burke and Fisher [17] is described in Section 3.3.

2.3 Error Recovery

Error recovery refers to the process of changing the internal state of a parser and/or the remaining input in order that parsing may continue.

No firm, commonly accepted categorisation of error recovery methods exist. The categories presented here broadly follow those of Grune and Jacobs [41], and Hammond and Rayward-Smith [42].

The three classes of error recovery methods suitable for linear-time parsers (such as those that use LL or LR methods) are *regional* error recovery, *local* error recovery, and *ad hoc* methods. The terms *regional* and *local* refer to the amount of surrounding context used to recover from an error.

Methods exist which use the entire input as the context. These *global* methods [2, 77] are inefficient and mostly used with general parsing algorithms such as Earley [30], CYK [49, 83], and Unger [75], which are themselves relatively inefficient compared to LL and LR parsing.

A number of published methods use a combination of error recovery schemes. These multi-level schemes have separate stages for error recovery. If one particular stage fails to find an error recovery, the next stage of error recovery is entered.

2.4 Ad hoc Error Recovery

Ad hoc methods are so-called because they cannot be automatically generated from a grammar, but instead rely on the ability of the parser writer.

They do not really form a class of error recovery. The three most common methods are *error productions*, *error tokens*, and *empty table slots*. Recovery methods using error productions and error tokens can also be legitimately classed as regional recovery schemes (Section 2.5). They are included in this section because they are not automatically derived from the grammar; they must be implemented manually by the parser writer.

2.4.1 Error Productions

Error productions are rules added to the grammar in anticipation of common syntax errors. The syntax error becomes part of the recognised language of the parser. Typically, if an error production is recognised by the parser, an appropriate error message is generated.

As an example, consider the following grammar which recognises a semicolon separated list:

```
SemiList  →  { Elements }
Elements  →  Elem
Elements  →  Elem ; Elements
```

The parser writer might anticipate that a programmer is likely to make an error by including a semicolon following the final item in the list. The introduced error production would look like:

```
Elements  →  Elem ;
```

A very specific error message can be given if an error production is matched, and parsing can easily continue. The main disadvantages are that only anticipated errors can be handled, and that modifying a grammar may make it ambiguous, or otherwise unsuited to whatever parsing method is being used.

2.4.2 Error Tokens

Error recovery by using *error-tokens* is used in *yacc* [46] and *bison* [23] and described by Aho and Johnsen in [1]. New rules are added to the grammar

that use a special token, the *error-token*[6]. When an error is detected, an error-token is inserted into the token stream prior to the token that caused the error. If a rule has been written that is able to recognise the error-token, then that rule is matched, and an appropriate error message can be given, and parsing can continue.

The method of matching rules containing error-tokens is usually extended in order for it to apply more generally. That is, if the state where the error occurred is not able to shift the error-token, then states on the stack are discarded until the state on top of the stack is able to shift the error-token. Finally, tokens are skipped from the input until a token can be shifted, and parsing resumes.

For example, consider the following grammar for recognising a function definition:

```
FunctionDef  →  Ident ( ParameterList ) StatementBlock
FunctionDef  →  Ident ( Error ) StatementBlock
```

In this example grammar, a parameter list is expected between the parentheses. If an error occurs while recognising a parameter list, the error-token 'Error' is inserted into the input before the token that cause the error. States on the stack are then popped until a state is found which can shift the error-token. Remaining tokens in the erroneous parameter list construct are discarded up until the closing parenthesis (the symbol following the error-token).

Error recovery using error tokens relies on the parser writer to augment the grammar with rules using error tokens, but can give reasonable quality repairs.

2.4.3 Empty Table Slots

In a parsing method which uses one or more tables to make parsing decisions, these tables are often sparse. The empty entries in the tables correspond to

[6] Not to be confused with the token that caused the error, also called the error token. When separated by a hyphen, 'error-token' refers to the special token described in this section. Without the hyphen, 'error token' refers to the erroneous token on which parsing halted.

no legal action for the parser, indicating an error in the input. This error recovery method associates an error handling function for each empty table slot that is invoked when that particular table entry is accessed.

This method has been used in a compiler for PL/C (a large subset of PL/I) [22], and recommended in [55]. Conway and Wilcox used a hierarchy of parse tables, greatly reducing the number of error entries that would have been present in a single table. Most of the actions for the error entries of the tables inserted, deleted, or replaced a symbol in order for the next symbol to be parsed.

This method requires a large amount of careful work from the parser writer. Few examples of this method exist because of the amount of work involved for the parser writer.

2.5 Regional Error Recovery

Regional error recovery methods attempt to identify and replace a phrase of the original input surrounding the point of error with a non-terminal that describes the phrase. Such techniques are also called *phrase level* error recovery.

2.5.1 Forward Moves

Leinius [53] develops an error recovery method that modifies the parse stack by performing an alternative reduction applied to a precedence parser. A *forward move* is used to gain information about the upcoming input; control is temporarily passed back to the parser to parse some of the remaining input. The scheme then uses the information from the parse ahead to find the shortest sequence of stack and input symbols that can be replaced by a non-terminal which gives a valid reduction and allows parsing to continue. A similar scheme for LR parsers is implemented by James [45].

Graham and Rhodes [40] present a scheme similar to Leinius [53], but with an initial backward move before the forward move. The backward move attempts to make further reductions on the stack. Costs are also assigned to the insertion and deletion of grammar symbols, so a least cost string of

symbols can be chosen as a modification to the stack. Levy [54] also uses a backward move and a forward move, but generalises it for any left-to-right parser. The backward move is used to find the last *beacon* symbol (a symbol, such as a left-parenthesis, which starts a significant syntactic construct) and the forward move is used to find and select a legal continuation.

Mickunas and Modry [60] present a scheme based on Levy and Graham and Rhodes extended for use with LR parsers. It restarts the parser on a forward move, then attempts a repair by inserting a single symbol. It requires remembering input already parsed to allow reconstruction of previous parse configurations.

Pennello and DeRemer [62] also present a scheme for LR parsers based on Graham and Rhodes, where the forward move is achieved by adding "recovery states" to the parser. The backward move is not used.

2.5.2 Costs

Vilares et al. [76] describe a method where a regional repair is found by searching the parsing automata of an LALR(1) parser. The method uses a bottom-up approach to calculate the possible repairs, and uses a cost mechanism to choose the repair.

2.6 Local Error Recovery

Local error recovery in the literature can be divided into methods which use an *acceptable-set* (two common forms of which are panic mode and follow sets) and those which repair the input. An acceptable-set is a set of tokens that must not be discarded from the input. All other tokens in the input are discarded until a token is found that is a member of the acceptable-set. The parser state is then changed to accept the symbol from the acceptable set, and parsing resumes. On the other hand, methods that repair the input attempt to insert and/or delete one or more tokens to correct the input.

2.6.1 *Panic Mode*

Panic mode is the simplest method of acceptable-set error recovery. Symbols in the acceptable set are static; they are determined by the compiler writer in advance and do not change during parsing. The symbols in the acceptable set (most commonly referred to as *fiducial symbols* or *marker symbols*) are typically those symbols in a language which mark syntactical constructs. Common examples are the semicolon, which often terminates (or separates) statements in a programming language, and an end-of-block marker such as '}' or 'END'.

When an error is detected, input symbols are deleted until a fiducial symbol is seen, then the parser adjusts its state until it can parse the symbol. For LL parsers, symbols of the prediction can be deleted, while for LR parsers, states are popped from the stack.

Aho and Ullman [4] present a simple algorithm for implementing panic mode for LL parsers. The adaptation of the algorithm for LR parsers is straightforward.

Panic mode can be implemented easily, but can often skip large portions of the input when searching for a fiducial symbol. In skipping chunks of input, other errors may also be skipped.

2.6.2 *Follow sets*

Follow sets are a more sophisticated form of panic mode. The symbols in the acceptable set are dynamic, determined by the context of the parse automatically by the parser, instead of statically determined by the parser writer.

Wirth [80] describes a Pascal compiler where he considers good syntax error handling is especially important because the language is to be used for teaching. The compiler uses recursive descent for syntax analysis, and for error recovery, *follow sets* are used, as discussed in Wirth [81] and Ammann [6]. Many variants exist, but the basic idea is to maintain a set of terminals (the follow set) which may follow the non-terminal currently being parsed. If an error occurs, input symbols may be discarded until a symbol is found that is a member of the follow set. An error message is given for the non-terminal where the error occurred, and parsing resumes assuming the current

non-terminal has been seen correctly, and the next input symbol is the one found to be a member of the follow set. Many variants exist [68, 61, 43, 66, 48, 11, 38, 74, 20, 18], mainly differing in how the symbols appearing in the follow set should be calculated. Hartmann [43] notes that this method works best if each symbol in the language is only used for one syntactic purpose. Pemberton [61] suggests removing highly overloaded symbols (such as commas) from the set of error recovery symbols as a way of improving the results. Such highly overloaded symbols have a low probability of being correct recovery points.

The follow-set error recovery method can be easily and automatically be added to recursive descent parsers, and is relatively efficient. It suffers from the same problems as panic mode, but to a lesser degree: Large chunks of the input can still be skipped, and errors are only fixed by discarding symbols, possibly leading to poor recovery to those errors that are better fixed by insertion.

2.6.3 Insertion-only (FMQ) method

Fischer, Milton, and Quiring [34] present a repair algorithm for LL parsers where costs are used to choose a repair using insert operations only. The method can only be used for grammars that are *insert-correctable*. An insert-correctable grammar has the property that every error can be repaired by using insertions only. That is, a grammar G is insert-correctable if, for every prefix x of a sentence in $L(G)$, and every terminal symbol a in G, there is a continuation of x that includes a.

They note that, even though the set of grammars where all possible errors can be repaired using only insertions (those that are *insert-correctable*) is a subset of LL(1) grammars, a grammar that is not insert-correctable can easily be modified to become so. It is sufficient (though not necessary) to augment a grammar with the rules $Z \rightarrow S$ and $Z \rightarrow SZ$, where S is the old start symbol, and Z is the new start symbol.

Anderson and Backhouse [8] note that the tables required for the insertion-only method can be much smaller and easier to calculate than the tables presented by Fischer et al. A similar method developed by Dion [29] is suitable

for LR parsing.

A recent paper by Kim and Choe [50] extends the FMQ algorithm by allowing insertion of non-terminals, similar to the way that Charles [19] uses in a single-transformation (Section 2.6.5) repair. The method is applied to an LALR parser, and incorporates validation.

The FMQ method has the advantage that no input is skipped, and a repair is always possible. Insertions are least cost, and costs may be fine tuned. The disadvantages are that some errors may be better repaired by deletion, the tables can be large, and transforming the grammar so that it is insert-correctable can be inconvenient.

2.6.4 Pattern matching method

Boullier [15] describes a pattern matching recovery method for LR parsers based on the method proposed by La France [37] for use in a bounded right context language [36] parser. The pattern matching method consists of matching a part of the input string containing the error onto one of a list of syntactically valid substrings. Recovery is made if a match is found, otherwise some other recovery method must be used. The mapping is performed by applying a number of transformational rules onto the input around the point of error. The set of syntactically valid strings is generated from the context information in the parse stack.

The syntactically valid strings in the set are all of the same length. Longer strings mean a larger probability of making a correct recovery, but also a larger probability of the input string containing more than one error. Boullier chooses an input string length of five symbols, including the error symbol, and the syntactically valid strings are four symbols in length. The set of syntactically valid strings, V, consists of the set of prefixes of all legal continuations of the correct input. Calculation of V is typically exponential in time, and frequently in space also.

Although La France lists 20 transformations that may be attempted when matching the input string with V, Boullier simplifies this to just three: insertion, deletion, or replacement of a single symbol. The simplification seems to give acceptable results, and improves efficiency.

Dain [25] re-invents the method for LR parsers. She uses the Fischer-Wagner algorithm [78] to compute the minimum distance between the input string and each syntactically valid continuation substring. The string with the lowest minimum distance is selected.

Tai [71] describes a method called pattern mapping which is used to choose a local repair to the input. The patterns model a transformation of one string into another, and have costs associated. A list of patterns is checked for a successful match with the unparsed input. The method is implemented into an SLR(1) parser. A technique appropriate for LL(1) parsers is developed by the same author [72], where costs of edit operations are used to choose a locally minimum-distance correction. The formal model used in the paper assumes errors occur in clusters separated by at least k correct symbols. The value of k is chosen by the parser writer.

This method can be flexible, and automatically generated by a parser-generator. It has the problems of being inefficient, recovery can be poor if errors occur very close to one another, and the recovery method will sometimes fail, requiring a backup recovery method to be used.

2.6.5 Single-transformation repairs

This simple error recovery scheme attempts to repair the input by a single transformation. Typically, these are:

- Insertion of a valid symbol before the error symbol
- Deletion of the error symbol
- Replacement of the error symbol with a valid symbol
- Merging the error symbol with the next symbol of input.

Because of the very limited range of errors that can be repaired, this method is always used in conjunction with another, often phrase-level, recovery method.

Burke and Fisher [17] use this method as the first stage in a three-stage error recovery scheme. A repair is considered valid if the next input symbol can be parsed.

Dain [26] also uses this method as the first stage of the "Recovery Method 1" described in the thesis. A repair must be able to parse a fixed number of input symbols successfully; the validation length may be greater than one. Merging is not implemented. The method, described in the context of LR parsers, falls back to a phrase level repair in the event the single-symbol repair fails.

Charles [18, 19] extends the method of Burke and Fisher by also allowing a non-terminal to be inserted before the error symbol, or to replace the error symbol. The work uses an LR parser, providing easy access to legal non-terminal moves (via goto edges in the parsing automata) from each state. Choosing among successful repairs is primarily done by choosing the repair that allows the largest number of symbols to be parsed following the error symbol.

2.6.6 Least-cost repairs

Least-cost repair methods attempt to restart parsing by transforming the erroneous input string into a syntactically valid input string. The transformation is acheived by a sequence of insertions, deletions, or replacements of input tokens. Each operation has an associated cost, with the cost of a transformation being the sum of the component operations. The transformation with the least cost is chosen as the repair.

Backhouse [10] describes a method for a recursive descent parser that chooses a local repair based on costs. The method is applied to the toy programming language PL/0 developed by Wirth [81]. Anderson et al. [9] propose a method based on the one in [10] where tables are created automatically that describe, for every possible combination of input symbol and state of the parser, what action should be taken to edit the input. The paper notes that, although this method produces better results than follow-set methods, it uses much more space for the storage of the tables— an Ada parser could not be generated, for example, because the space requirements exceeded the memory of the machine they were using at the time (1983). Also noted in the paper are two main deficiencies of locally least-cost error recovery.

- The local nature of the choice means a second error could be generated

by the repair of the first. For example, the input 'a := 2 c' (for a Pascal-like language) has an error at symbol 'c'. Two valid repairs are inserting a semicolon or operator before the error symbol. If the symbol following the error symbol was, say, ':=', then inserting an operator would be a poor repair, resulting in another error, but inserting a semicolon would allow parsing to continue further. Later papers [57] use the idea of a validation length to overcome this limitation: A repair is not used unless a number of upcoming input symbols can be successfully parsed. Backhouse's method effectively has a validation length of one.

- The actual error may be prior to the error detection point. That is, the best repair is obtained by changing the input that has already been parsed successfully. This deficiency is unavoidable due to the definition of *locally* least-cost recovery. In practice, such errors are rare.

Backhouse's work generalises most of the other local recovery methods: panic mode, follow-set method, the insertion-only method by Fischer et al., and Boullier's pattern matching method. The single-transformation repair can be thought of a subset of least-cost repair, although merging two tokens is not usually supported in least-cost methods. Table 2.1, based on a similar table in [42], shows the parameters necessary to emulate the other local recovery methods.

Anderson and Backhouse [7] apply the locally least-cost method to the Earley parsing algorithm [30]. Clare et al. [21] describe a similar technique for LALR(1) parsers and note that a validation length of one produced poor repairs. Implementation and performance details in the paper are sketchy, but memory usage was noted to be excessive for non-trivial repairs.

Fischer and Mauney [33] extend the FMQ algorithm (Section 2.6.3) for LL(1) parsers with deletions, making it similar to the least-cost repair of Backhouse [10] but without replacements. They also modify the FMQ algorithm so that tables are computed as needed rather than pre-computed, and they require a validation of upcoming symbols for a successful parse. They note that the combination of validation and least-cost repair results in good quality repairs.

panic mode:	set all insert and replace costs to ∞ set all synchronising symbol delete costs to ∞ set all other delete costs to 1
follow-set:	set delete costs of all symbols in follow-set to ∞ set all other delete costs to 1 set insert costs to appropriate values set replacement costs to ∞
pattern-matching:	set delete costs to 1 set appropriate insert and replace costs to 1 set other costs to ∞
insert-only:	set all delete and replacement costs to ∞ set all insert costs to finite values
single-transformation:	set insert, delete, and replacement costs to 1 restrict potential repairs to one operation

Table 2.1: Configuration of costs Backhouse's least-cost recovery method to simulate or duplicate other recovery methods

McKenzie, Yeatman and de Vere [59, 82] present an algorithm for LALR(1) parsers similar to the one by Fischer and Mauney [33] for LL(1) parsers. It uses the information in the parsing automata to find the least-cost recovery; no tables need be pre-calculated. Although the repairs that are produced are of good quality, there is no upper bound on the time to find a recovery. More information on this method is in Section 3.1. A pruning mechanism is implemented by McKenzie et al. that prevents cycles in the parsing automata being traversed. Bertsch and Nederhof [14] show that this is too aggressive, sometimes pruning options which would lead to a least-cost repair. They develop a safe, but slower, variant of the pruning technique.

Kim and Choe [50] present an algorithm which they claim is more efficient than that of McKenzie et al. [59]. They transform the problem of finding the least-cost repair in an LR parser to a graph search in an LR machine. A shortest-path algorithm is then used to find the least-cost repair. Tests were conducted using correct programs (collected randomly from the internet) which were subsequently corrupted with single token errors. Such simple errors are likely to give good results on even poor repair algorithms. One

example of a multiple token repair is presented, but its context is very narrow, the repair consisting of a number of right parentheses. No mention is made of how costs were assigned, and deletions are not explicitly considered.

A recent paper by Corchuelo *et al.* [24] presents an almost identical algorithm to McKenzie's method, with the addition of being able to shift input tokens during repair, but appear unaware of McKenzie's work. As an example of shifting tokens during a repair, the input '(1 2 $' to a parser recognising expressions, may result in a repair of inserting '+', shifting '2', and inserting ')'. In order to restrict the number of configurations, they set upper limits on the number of insertions, the number of deletions, and fix the size of the region of input over which repairs will take place. Panic mode is selected as the fall-back mechanism if a repair is not found. Searching for repairs is done breadth-first, as opposed to the more efficient method of McKenzie, which uses a priority queue ordered by the cost of repair. Loops are not traversed, resulting in the same error of McKenzie's pruning method.

Chapter III

Framework

This chapter describes the framework within which the work presented in this thesis lies.

The pruning algorithms developed in this thesis apply to any locally least-cost error repair algorithm for an LR-based parser, but have been tested with, and compared to, McKenzie's [59] algorithm. The pruning algorithms reduce the search space while maintaining the property of finding least-cost transformations.

This chapter describes McKenzie's [59] algorithm to which the pruning algorithms presented in this thesis were added, including a description of a more efficient priority queue implementation developed in the course of this thesis.

McKenzie's algorithm was added to the *bison* parser generator for the experimental analysis in Chapter 8. Section 8 explains the solution used to overcome the lack of immediate error detection in LALR parsers.

3.1 McKenzie's algorithm

McKenzie's algorithm finds a locally least-cost repair by means of *configurations*. A *configuration* is a potential repair; it encapsulates the state of the parser as a result of a sequence of insertions and/or deletions. Each configuration is a 4-tuple (S, I, D, C) where:

- S denotes the stack of states.
- I is a list of symbols inserted at the point of error.

- D is a list of symbols deleted from the remaining input at the point of error.

- C is the combined cost of the insertions and deletions.

When an error occurs, the repair algorithm (Listing 3.1) is started with a configuration containing the stack at the point of error. The configuration is placed in a priority queue ordered by C, the combined cost of the insertions and deletions made to the remaining input. A loop is entered that removes the lowest cost configuration from the priority queue and checks to see whether a valid repair is possible using that configuration. If there is no possible repair, new configurations are generated from the current configuration by following the shifts and reductions from the state on top of the current configuration's stack.

Listing 3.1: Pseudo-code outline for the algorithm described by McKenzie et al.

```
1  config c
2  priority queue Q [Ordered by cost of config]
3
4  [configuration 4-tuple elements are (parse stack, symbols inserted, symbols deleted,
   and cost), referenced as c.S, c.I, c.D, c.C]
5  c = (<parse stack at point of error>, [], [], 0)
6
7  Q.insert(c)
8  while true:
9      c = Q.head()
10     if repair-possible(c):
11         break
12     else:
13         foreach shift s to state t from c:
14             c' = ( c.S + t, c.I + s, c.D, c.C + I(s) )
15             Q.insert(c')
16         foreach reduction A → α from c:
17             c' = ( c.S − |α| + goto(A), c.I, c.D, c.C )
18             Q.insert(c')
19         [Make new config by deleting next input token]
20         c' = ( c.S, c.I, c.D+ <next-token>, c.C + D(<next-token>) )
21         Q.insert(c')
22 Restart parsing with repair-config c
```

The three pruning algorithms contributed in this thesis modify the algorithm in Listing 3.1 by restricting certain shifts or reductions from adding new configurations to the priority queue.

3.2 $O(1)$ priority queue improvement

One of the contributions to this thesis is a specialised priority queue, suitable for use in McKenzie's algorithm, that has time complexities of $O(1)$ for both the insert operation and the remove-min operation. It is a specialised version of the calendar queue, described by Brown[16] in the context of representing the pending event set in a discrete event simulation.

The typical undergraduate data-structures text book, such as Kingston [51] suggest a number of different tree structures for the implementation of a general priority queue, such as a Heap, 2-3 tree, or Fibonacci heap. These implementations may be appropriate for entries where a comparison between two entries (resulting in a less-than, equal-to, or greater-than relationship) is the only information known. In the case of McKenzie's algorithm, however, more information is known about the possible entries that allows a more efficient algorithm to be devised.

In particular, the extra information available regarding the entries in the priority queue is summarised as:

- The value of any inserted entry will not be less than the current minimum entry. We know this because no tokens are permitted to have a negative insert or delete cost.

- The value of any inserted entry will always be less than a constant n plus the value of the current minimum entry, where n is the largest insert or delete cost (There is only ever one insert or delete at a time).

- n can easily be made relatively small; it is the cost of the most expensive symbol operation. Restricting the most expensive insert or delete to a cost of 50 does little to restrict the combinations of costs that are useful.

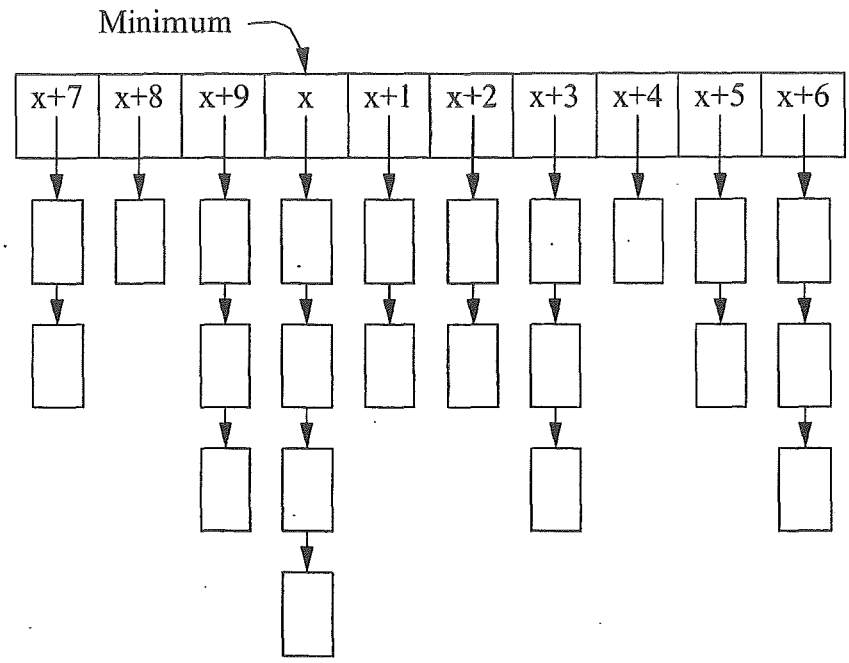

Figure 3.1: Priority queue implementation with $O(1)$ for both insert and remove-min operations

- The values of entries are discrete (Integers).
- The number of entries is often much greater than n.

It follows from the points above that the maximum number of different values of entries present in the priority queue is equal to $n + 1$ (the number of different costs from 0 to n inclusive). The priority queue, then, can be implemented by an array of length $n + 1$, the elements of which point to a linked-list of entries having the same value. Figure 3.1 shows an example where $n = 9$, and the value of the minimum entries in the table is x.

The remove-min operation removes the first entry of the linked list at the minimum position in the array. If no elements exist (the linked list is empty), the minimum index is incremented modulo $n + 1$ until a non-empty entry is found.

The insert operation of an entry with cost y is done by calculating the difference between y and x (the cost of the minimum entry in the priority

queue), and adding the difference to the current minimum index modulo $n+1$. Insert index i for an entry with cost y is given by:

$$i = (minimum + (y - x)) \mathrm{mod}(n+1)$$

thus the insert operation is also $O(1)$.

3.3 The bison parser generator

McKenzie's algorithm, along with the three pruning algorithms presented in this thesis, were added by the author to the *bison* parser generator. *Bison* produces an LALR(1) table with default reductions. It therefore has the correct prefix property (parsing will halt before shifting an erroneous input symbol), but does not have the immediate error detection property (extra reductions may be performed if the next symbol is not legal) (Section 2.2).

The usual technique used[1] to solve this is the *deferred parsing* of Burke and Fisher [17]. Another approach [19, 39] involves generating extra information in addition to the LALR(1) tables indicating the lookahead symbol for each reduction, effectively negating the benefit (smaller parse tables) of using default reductions in the first place.

Deferred parsing makes use of two parsers. The first checks for syntactic correctness only, and does not perform any semantic actions with reductions. The second parser, which is k tokens ($k \geq 0$) behind the first, always has correct input, and performs semantic actions with reduce actions. Only the case $k = 0$ need be considered for useful detection of error configurations. For $k = 0$, both parsers shift tokens simultaneously. The first parser then performs a sequence of reductions if necessary then checks if the next symbol is able to be shifted. If so, then the second parser also does the reductions, and parsing continues. If not, an error has occurred, and the correct configuration of the parser to begin a repair is given by the second parser. The deferred parsing technique, as implemented by McKenzie et al., imposes a speed penalty of approximately 40% on correct programs.

[1] A number of papers fail to mention how they solve this problem, or whether the problem was given any thought at all.

The deferred parsing technique for the case $k = 0$ can be thought as a single parser that 'splits' into two parsers after each shift, and then rejoins into one parser before the next shift. A more efficient variant of deferred parsing, discovered in the course of this thesis by the author, was implemented into *bison*. The technique works by only splitting the parser in those states where an error in the next input symbol would result in an incorrect parser configuration if no splitting occurred. In particular, no splitting is required in states where the only action is a single reduction. Typically, for programming language grammars, 50%–70% of states with a reduction action have only a single reduction (and no shifts) and the parser need not be split. With the more efficient deferred parsing technique described, the speed overhead for parsing correct programs shrinks from approximately 40% to approximately 5%.

Chapter IV

Algorithm *FolNonTerm*: Following non-terminals

This chapter introduces the *FolNonTerm* algorithm developed as part of the thesis. The algorithm modifies the search for a repair by following non-terminals as well as terminals. Following non-terminals in the search allows many reductions to be pruned.

4.1 Motivation

To see the motivation behind following non-terminals, it is useful to look at a repair using McKenzie's [59] algorithm from Listing 3.1. For the example, the following CFG:

```
S  →  'e' A 'e'
A  →  '=' | '-=' | '+='
```

produces the parser shown in Figure 4.1.

The terminal symbols are assigned the following costs:

$$C_i(\text{'+='}) = 2$$
$$C_i(\text{'-='}) = 2$$
$$C_i(\text{'='}) = 1$$
$$C_i(\text{'e'}) = 3$$

For simplicity, we will assign the delete cost of a symbol to be the same as its insert cost. If the erroneous input e$ was then given to the parser (the '$' symbol denotes end of input), an error would occur in state 1 because there is no shift on $ from that state. From state 1, three possible symbols (+=, -=, and =) are considered for insertion. The priority queue of configurations might then look like this:

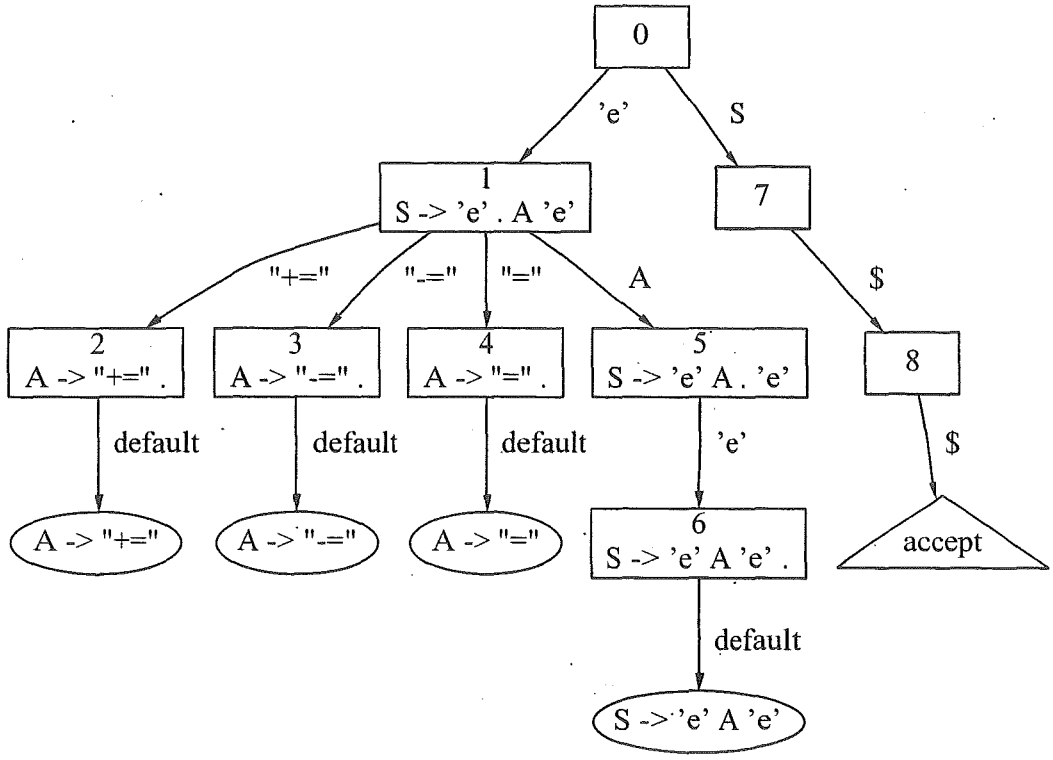

Figure 4.1: Example parser showing motivation for the *FolNonTerm* algorithm

(∇014 , = , ϵ , 1)
(∇012 , +=, ϵ , 2)
(∇013 , -=, ϵ , 2)

The searching algorithm then removes the configuration at the head of the priority queue, and checks whether parsing can be restarted using that configuration. In this case, there is only a reduction from state 4. Performing the reduction results in a configuration that has the same cost (no symbols were inserted/deleted) and is placed into the priority queue:

(∇015 , = , ϵ , 1)
(∇012 , +=, ϵ , 2)
(∇013 , -=, ϵ , 2)

The new configuration is at the head of the priority queue, and is now removed and checked to see whether parsing can be restarted using it. Parsing cannot be restarted, but the e symbol can be inserted, generating a new configuration (∇0156 , = e , ϵ , 4) that is put onto the priority queue.

The priority queue now looks like this:

(∇012 , += , ϵ, 2)
(∇013 , -= , ϵ, 2)
(∇0156 , = e , ϵ, 4)

Neither of the two configurations at the head of the priority queue is able to restart the parse, but both can do a reduction, and create a new configuration with a stack of ∇015. Any further configurations resulting from these two configurations can never produce a least-cost repair because a lower cost configuration has already had a stack of ∇015 (the stack alone determines the possible strings of tokens that may be parsed). The two higher-cost configurations can be safely pruned. An alternative (but simpler) way of achieving the same result is to insert non-terminals but not do any reductions. As well as the three shifts from state 1, we would also create a

configuration for the goto: (∇015 , A , ϵ , 1). The cost of inserting a non-terminal is the least-cost insert string that can be generated by that non-terminal (the string = for A). Now, although the other three configurations will still be tested to see whether parsing can be restarted, the configurations resulting from the reductions can be safely pruned. This is the method the *FolNonTerm* algorithm uses.

Not all reductions are pruned. The exception is when a reduction would take the stack back to a point that no previous configuration has yet searched. For the example above, a configuration with a stack of ∇0156 would still have required the reduction back to state 0 and forward to state 7 because state 0 had not yet been searched (searching for a repair began at state 1, the error state).

4.2 Description

The *FolNonTerm* algorithm involves following gotos from a state as well as shifts and reductions. This initially increases the number of configurations required, rather than reducing it. Following non-terminals has the effect of inserting a string of symbols at a time (the minimum cost insert string for that non-terminal) and allows us to do fewer reductions: A reduction involves popping a number of states off the stack, followed by a goto that pushes a state onto the stack. After popping some states off the stack, we may prune the reduction if the state reached has already been searched (we call this a *close-reduce*). In this case, the goto that we would have performed as part of the reduction would have already been performed earlier by following that non-terminal. Other reductions— those that pop back so far into the stack that we haven't searched from that state yet— must be followed (we call this a *far-reduce*).

Distinguishing between a close-reduce and a far-reduce requires an additional element in a configuration, the *base-stack*. The addition of the base-stack makes a configuration a 5-tuple.

Listing 4.1: The McKenzie algorithm augmented with the *FolNonTerm* pruning algorithm

```
1  config c
2  priority queue Q [Ordered by cost of config]
3
4  [configuration 5-tuple elements are (parse stack, base stack, symbols inserted, symbols deleted, and cost), referenced as c.S, c.Z, c.I, c.D, c.C]
5  c = (<parse stack at point of error>, c.S, [], [], 0)
6
7  Q.insert(c)
8  while true:
9      c = Q.head()
10     if repair-possible(c):
11         break
12     else:
13         foreach shift s to state t from c:
14             c' = ( c.S + t, c.Z, c.I + s, c.D, c.C + I(s) )
15             Q.insert(c')
16         for each goto g to state t from c:
17             c' = ( c.S+t, c.Z, c.I+least-cost-string(g), c.D, c.C+I(g))
18             Q.insert(c')
19         for each reduction A → α from c:
20             if not canprune_FolNonTerm(c, |α|):
21                 c' = ( c.S − |α| + goto(A), c.I, c.D, c.C )
22                 c'.Z = c'.S
23                 Q.insert(c')
24         [Make new config by deleting next input token]
25         c' = ( c.S, c.Z, c.I, c.D+<next-token>, c.C + D(<next-token>))
26         Q.insert(c')
27 Restart parsing with repair-config c
28
29 proc canprune_FolNonTerm(config c, reduction-length r):
30     n = length(c.S)−length(c.Z)
31     if r > n:
32         return false
33     else:
34         return true
```

4.3 Implementation notes

The run-time overhead of the *FolNonTerm* algorithm is minimal. Goto actions can be easily read from tables, although *bison*'s method of compressing the parse tables means that the information cannot be extracted [82]. A separate table was constructed with the goto actions from each state. The goto-table is relatively small compared to the existing tables in the *bison* parser runtime environment.. A simple implementation (with no compression) increases the size of the tables by about 40%. Deciding whether to prune a reduction is a simple comparison of the length of the reduction and the size of the stack since the start of the repair algorithm (or the last far-reduce).

Some extra space is also required during a repair because of the additional storage of a base-stack in the configuration. However, the base-stack is a prefix of the parse-stack, so only an index into the parse-stack is required, resulting in minimal extra run-time space overhead.

Chapter V

The *LoopLimit* algorithm

This chapter introduces the *LoopLimit* algorithm, developed as part of the thesis. It restricts the search for a repair by limiting loops in the generated parser when there is no possibility that following those loops will lead to a least-cost repair.

After some examples are presented illustrating the problem of loops, it is shown how loops in the parsing automata are related to constructs in the CFG the parsing automata was derived from. The calculation is presented of the maximum number of times a loop must be traversed to ensure least-cost repair. Looping beyond the maximum never results in a least-cost repair. Finally, some implementation details are presented, and implementation assumptions justified.

5.1 Motivation

While searching for a repair, particularly a longer repair, it was found that many of the possible configurations generated were the result of traversing a loop in the parsing automata a number of times. Often these configurations were entirely useless— any valid repair resulting from them would also be found at a lower cost by traversing the loop fewer times.

There are some loops in a parsing automata that we must traverse when searching for a valid repair, but only a limited number of times, and others that we need not traverse at all (a minimum of 0).

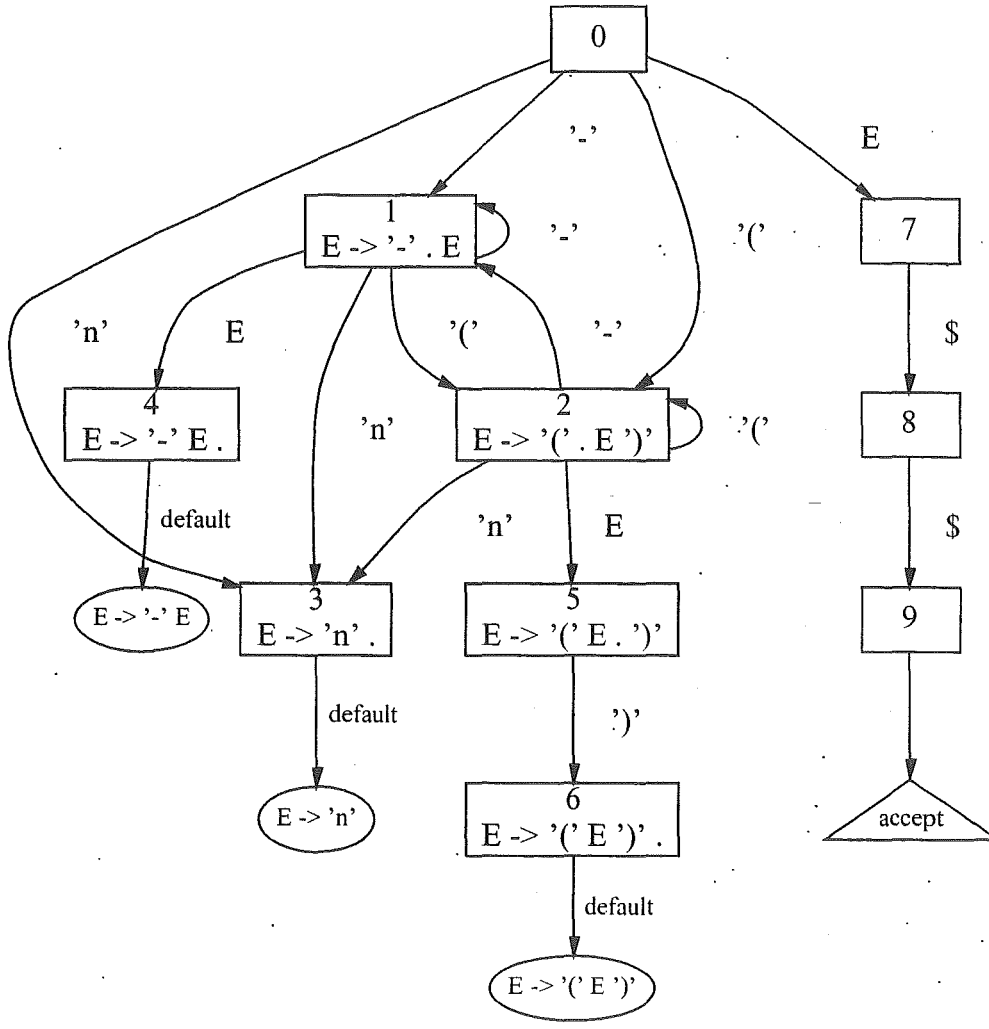

Figure 5.1: Example parser showing motivation for the *LoopLimit* algorithm

5.1.1 *Loop example*

The CFG below, describing a subset of expressions, is used to produce the parsing automata shown in Figure 5.1

$$\begin{aligned} \texttt{E} &\rightarrow \texttt{`(' E `)'} \\ \texttt{E} &\rightarrow \texttt{`-' E} \\ \texttt{E} &\rightarrow \texttt{`n'} \end{aligned}$$

Using the input string -)$, after shifting the first symbol, an error occurs in state 1, because there is no shift from state 1 on a) symbol. The recovery function is called (we will use the McKenzie repair-search algorithm shown in Listing 3.1), and the search for a valid repair begins.

There are four edges (three shifts and one goto) from state 1 that can be searched. These possibilities are entered into a priority queue sorted by the cost of the repair. For simplicity, we will assume here that the insert and delete cost for each symbol is 1. The insert cost for a non-terminal (in this case, there is only one non-terminal, E), is the insert cost for the minimum cost insert string that can be derived from that non-terminal. In this case, the rule $E \rightarrow n$ provides the minimum cost insert string, n, of cost 1. The priority queue would initially look like this (possibly in a different order):

∇110) , - , ϵ , 1
∇210) , (, ϵ , 1
∇310) , n , ϵ , 1
∇410) , E , ϵ , 1
∇10 , ϵ ,) , 1

Each 'configuration' in the queue is a four-tuple. The first element is a pair indicating the state-stack and remaining input for this configuration. The next three elements are the insert-string, delete-string, and total cost for this configuration.

The first configuration above, generated by inserting a "-", will never result in a repair that is of lowest cost. Any valid repair that is generated by initially inserting a "-" can also be generated without inserting it.

The second configuration listed above brings us to state 2. From state 2 there are also four edges to be searched, and like state 1, there is a looping edge from state 2 back to itself. This looping edge must be followed multiple times. Failure to do so could result in repairs not being found. For example, if the remainder of the input was "))))", then the loop would have to be followed four times to match the remaining input (the first right parenthesis is matched by the initial shift to state 2).

5.1.2 Exponential example

In certain circumstances, the number of strings derivable from a looping portion of the parsing automata is exponential in the length of the string.

For example, consider the grammar:

$$\begin{aligned} \text{S} &\rightarrow \text{'(' S ')'} \\ \text{S} &\rightarrow \text{'[' S ']'} \\ \text{S} &\rightarrow \epsilon \end{aligned}$$

which produces the parsing automata shown in Figure 5.2. From Figure 5.2 we see that from either state 1 or state 2, the set of continuations $t_1 t_2 \ldots t_n$ must contain every pair of strings where t_i ='[' and t_i ='('. For a set of continuations of length n, there are 2^n possible combinations.

This construct occurs in many programming languages. For example, a programming language with arrays and expressions has this exponential explosion problem: an expression may be parenthesised and contain an array reference, and an array reference contains an expression.

5.2 Justification

The following sections show that all non-left recursion ($A \rightarrow^* \alpha A \beta, \alpha \neq \epsilon$) results in a grammar results in a loop in the corresponding parsing automata. Furthermore, by showing that left-recursion and non-recursion do not cause loops to appear in the parsing automata, we can say that all loops in the parsing automata are caused by non-left recursive rules in the grammar. Section 5.2.4) describes the minimum number of times a loop must be traversed to be sure that all possible least-cost repairs will be found.

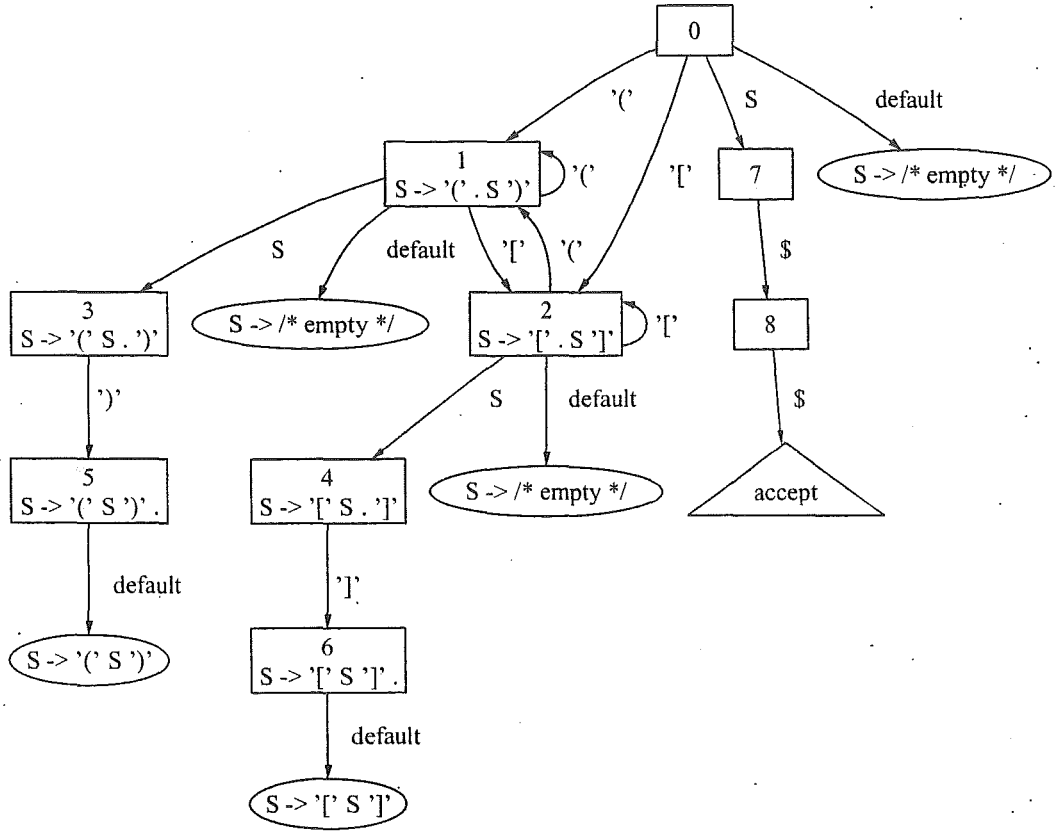

Figure 5.2: An example of parsing automata exhibiting exponential growth in the number of possible insert strings resulting from the interaction of loops.

5.2.1 *Non-left recursion results in loops*

If we have a non-left recursive rule:

$$A \rightarrow \alpha_1\alpha_2 \ldots \alpha_n A\beta, n \geq 1, \alpha_i \in (N \cup T), \beta \in (N \cup T)^*$$

then we know that at some point in the construction of the parsing automata, the item $A \rightarrow \cdot\alpha_1\alpha_2 \ldots \alpha_n A\beta$ will appear. Let this state be s_0. The relevant states will contain at least those items shown in Table 5.1. The loop occurring in this example is the path $s_1, s_2, \ldots, s_n, s_1$. In some cases, due to 'interference' from other rules, the core item(s) resulting from state s_n on α_1 may not be identical to the core items in state s_1. If this is the case, a new state s_{n+1} is created with the same item as s_1 for the non-left recursive rule. State s_{n+1} may shift/goto state s_2 on α_2 if the corresponding core items are identical, otherwise a new state is created in the same manner as above. New states are created as necessary, until, at some point, a possible new state contains the same core items as a previously seen state. At this point, instead of creating a new state, the shift/goto will be to the previously seen state.

State	Items	Actions
s_0	$A \rightarrow \bullet\, \alpha_1\alpha_2 \ldots \alpha_n A\beta$	shift/goto s_1 on α_1
s_1	$A \rightarrow \alpha_1 \bullet \alpha_2 \ldots \alpha_n A\beta$	shift/goto s_2 on α_2
$\vdots$	$\vdots$	$\vdots$
s_{n-1}	$A \rightarrow \alpha_1\alpha_2 \ldots \bullet\, \alpha_n A\beta$	shift/goto s_n on α_n
s_n	$A \rightarrow \alpha_1\alpha_2 \ldots \alpha_n \bullet A\beta$ $A \rightarrow \bullet\, \alpha_1\alpha_2 \ldots \alpha_n A\beta$	goto s_m on A shift/goto s_1 on α_1

Table 5.1: State table resulting from the rule $A \rightarrow \alpha_1\alpha_2 \ldots \alpha_n A\beta$

This proof also holds for a set of production rules which together exhibit non-left recursion. Consider the set of rules:

$$\begin{aligned} A_0 &\rightarrow \omega_0 A_1 \beta_0 \\ A_1 &\rightarrow \omega_1 A_2 \beta_1 \\ &\vdots \end{aligned}$$

$$A_k \rightarrow \omega_k A_0 \beta_k$$

where $|\omega_0 \ldots \omega_k| \geq 1, \beta_n \in (N \cup T)^*$.

When an item reaches the non-terminal of the next rule, e.g.:

$$A_j \rightarrow \omega_j \bullet A_{j+1} \beta_j$$

the state that the item is in will have a shift or goto on the first symbol of A_{j+1}. In this way, all rules in the set are 'chained' together, with the last one chained to the first in the same manner as in table 5.1.

5.2.2 Left recursion result in no loops

If we have a left recursive rule:

$$A \rightarrow A\beta \qquad\qquad \beta \in (N \cup T)^*$$

then at some point the item $A \rightarrow \bullet A\beta$ will appear in the construction of the parsing automata. Let this state be s_0. The states resulting from the left recursive rule will contain at least those items represented in Table 5.2. Thus there are no loops in the automata caused by left recursion.

State	Items	Actions
s_0	$A \rightarrow \bullet A\beta$	goto state s_1 on A
s_1	$A \rightarrow A \bullet \beta$	shift/goto state s_2 on β
s_2	$A \rightarrow A\beta \bullet$	reduce: $A \rightarrow A\beta$

Table 5.2: State table resulting from production $A \rightarrow A\beta$

5.2.3 Non-recursive rules result in no loops

If we have a non-recursive rule:

$$A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n \qquad\qquad \alpha_i \in (N \cup T)$$

then at some point the item $A \rightarrow \bullet \alpha_1 \alpha_2 \ldots \alpha_n$ will appear in the construction of the parsing automata. Let this state be s_0. The states resulting from the left recursive rule will contain at least those items represented in Table 5.3. Thus there are no loops in the automata caused by non recursive rules.

State	Items	Actions
s_0	$A \rightarrow \bullet \alpha_1 \alpha_2 \ldots \alpha_n$	shift/goto s_1 on α_1
s_1	$A \rightarrow \alpha_1 \bullet \alpha_2 \ldots \alpha_n$	shift/goto s_2 on α_2
$\vdots$	$\vdots$	$\vdots$
s_n	$A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n \bullet$	reduce: $A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$

Table 5.3: State table resulting from production $A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$

5.2.4 Number of times a loop must be traversed in a repair

The pumping lemma [44, 70] for context free grammars states that, for a context-free language L, there is a constant n, depending only on L, such that if z is in L and $|z| \geq n$ then we may write $z = uvwxy$ (letters $uvwxyz$ represent strings of 0 or more terminals) such that:

1. $|vx| \geq 1$,
2. $|vwx| \leq n$, and
3. for all $i \geq 0$, uv^iwx^iy is in L.

The sequence v^iwx^i in z is produced by two derivations of a non-terminal: $A \rightarrow^* vAx$ and $A \rightarrow^* w$. If $|v| = 0$ then we have a left-recursive rule. If $|x| = 0$ then we have a right-recursive rule. If both $|x| \neq 0$ and $|v| \neq 0$ then we have a middle-recursive rule.

For a left recursive rule (uwx^iy), assume an error occurs somewhere in uw, and is repaired up to the end of uw. There are n symbols in x: $x_1x_2 \ldots x_j \ldots x_{n-1}x_n$ If the successful repair is to be found at x_j, then only the symbols up to x_j need to be inserted. If there is no successful repair found at any x_j, then the symbols in x need not be inserted again after x_n

(uwx^2y), as they will not find a valid repair. The successful repair will be found in y. It is not necessary to follow left recursion for $i \geq 2$ to find the least-cost repair.

In the same way it is not necessary to follow right recursion for $i \geq 2$.

Middle recursion (uv^iwx^iy) must be followed during a repair. Assume an error occurs somewhere in u, and is repaired up to the end of u. We do not know whether the successful repair will be found in v^i, w, x^i or y. A 'successful' repair is defined as a repair where parsing is permitted to continue for a specified minimum number of tokens, called the validation length l. If the successful repair is in w or y, then no symbols of v or x need to be inserted. This cannot be determined in advance, however. If the correct repair is in x^m where $m \leq l$, then v^m must be inserted. The worst case is when $|x| = 1$ and the successful repair (i.e. we have found l tokens which we can now parse successfully) is found at x^0, requiring v^l to be inserted. In general, the maximum number of v's we must insert for middle recursion is given by $\lceil \frac{l}{|x|} \rceil$.

5.3 *Algorithm Description*

Broadly speaking, as a new state is placed on the parse stack (when a new token is inserted) a check is performed to see if this state increases the number of loops that have been done so far. If that number is increased beyond the validation length, then the configuration is useless (that is, it will never result in a least-cost repair).

There are some simplifying assumptions that have been made for the implementation.

- Loops arising from right-recursive rules are counted in the same way as loops arising from middle-recursive rules, even though a right-recursive derived loop need only be traversed a maximum of once. This simplification was made because it is difficult in a complex grammar to accurately distinguish whether a given loop is a result of a right recursive rule only. Least cost repairs will still be found with this simplification, although it would most likely take more time than if a reliable way to

differentiate the loops was available.

- For a similar reason as the previous point, a middle recursive loop ($A \rightarrow \alpha A \beta$ is assumed to have $|\beta| = 1$. This simplification requires loops to be traversed the same number of times as the validation length. Least cost repairs will still be found with this simplification.

- The number of loops is determined by a count starting at the first state of the stack after the error, and discards any overlapping loops. This means that in a few cases, a stack such as $1, 2, 3, 1, 3, 2$ would be seen as having one loop $(1, 2, 3, 1)$, rather than two loops $(3, 1, 3$ and $2, 3, 1, 3, 2)$. While it is certainly possible to find the maximum number of loops in a stack segment, no efficient algorithm was found to do so[1]. Also, preliminary evidence indicates that the probability of a stack occurring where counting from the first state results in a fewer loops than counting from subsequent state is very low, certainly low enough that the overhead of maintaining a maximum count for a stack would outweigh the cost of the occasional missed loop when counting from the first state. Least cost repairs will still be found with this simplification

- The validation length will be assumed to be the maximum number of times that a loop must be traversed. This is not always true. For example, the following (rather contrived) grammar:

```
S  →  'a' S 'b'
S  →  'a' 'a' S 'c'
S  →  'a'
```

produces the parsing automata shown in Figure 5.3. Let us assume a validation length of three for this example. If this parser is fed the (incorrect) input string "c c c", an error occurs immediately. To recover from that error, the token string "a a a a a a" must be inserted (assuming the delete cost of "c" is more than twice the insertion cost of a

[1] An $O(n^2)$ algorithm is found by starting the loop count at each position in the list, and taking the maximum. It is conjectured that an $O(n \log n)$ algorithm exists.

"a"). Inserting the tokens "a a a a a a" requires traversing the loop four times— once more than the validation length of three.

This construct is very unlikely to appear in a typical programming language. Middle recursive rules are typically used in programming languages for constructs that must be nested. For example blocks in C/Java are delimited by "{" and "}", and expressions can be nested with parentheses. It would make little sense from a language design point of view to have one closing delimiter ("b" in the example) matching one opening delimiter ("a") and a different closing delimiter ("c") matching *two* of the same opening delimiters ("a a").

Listing 5.1: The *LoopLimit* pruning algorithm

```
1 proc canprune_LoopLimit(config c, edge e):
2     st = stack generated by following edge e from c
3     if count_loops(st) > v:
4         return true
5     else:
6         return false
```

While the algorithm in listing 5.1 appears simple, there is some complexity hidden in the "`count_loops`" function. Before describing how (and which) loops are counted, we first define what is meant when we use the term 'loop'.

A loop is defined as a path of vertices $v_1, v_2, \ldots, v_n$ in the parsing automata where $v_1 = v_n$ (a cycle). The vertex v_1 is not permitted to occur anywhere else in the loop, although it is permitted that a loop starting from v_1 completely contains another loop starting from v_i. For example, the sequence $0, 1, 4, 6, 9, 6, 3, 1$ contains two loops: $6, 9, 6$ and $1, 4, 6, 9, 6, 3, 1$.

5.3.1 Counting loops

Only those loops that arise from inserting tokens should be counted; loops that begin in the stack before the error was encountered must not be counted. For example, if a Java-like language has a token sequence that looks like "{ { { <error>)) } } }", then, although at least three loops have already been encountered (as a result of a statement rule looking like S → {

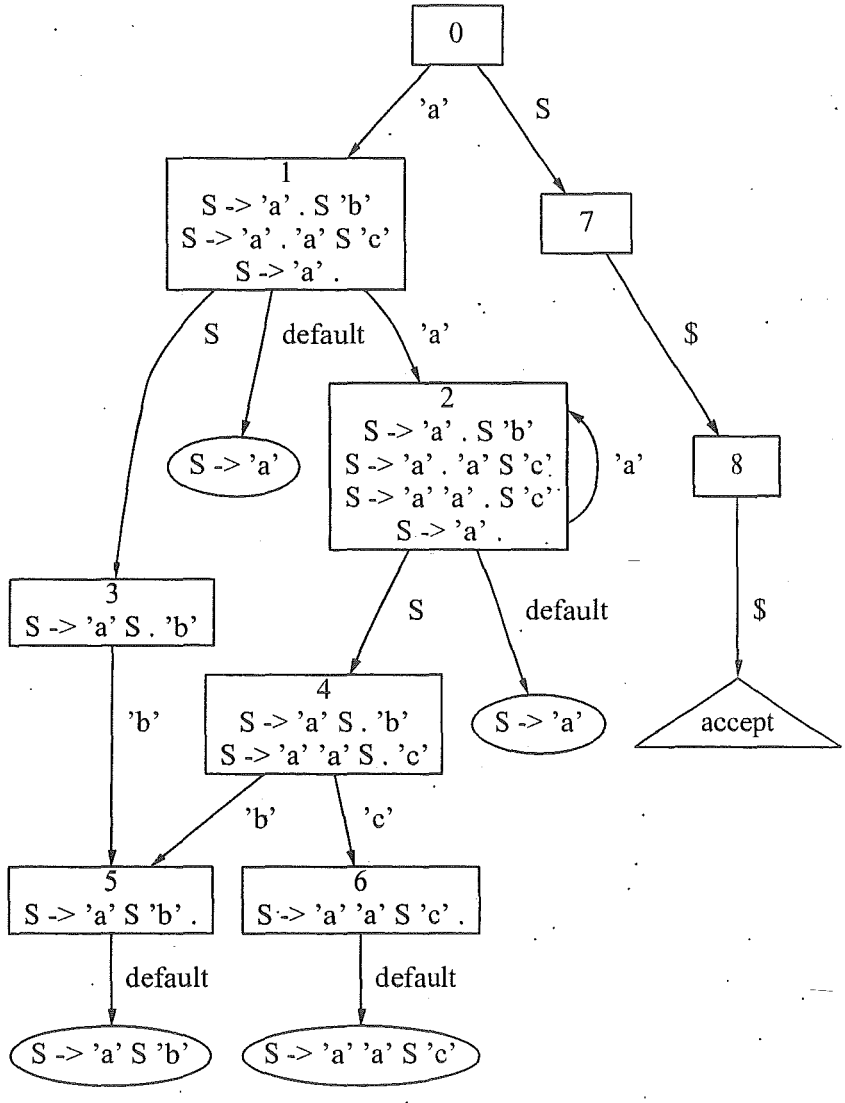

Figure 5.3: Example parser showing a loop that must be traversed more often than the validation length

S }), a fourth loop (resulting from an expression rule like E → (E)) would have to be traversed to provide the matching left parentheses for the two existing right parentheses.

Overlapping loops

When counting loops in a portion of the stack, no two loops may be partly overlapping in terms of their position in the stack, apart from a possible common boundary. That is, for two loops to be counted, they must be either disjoint (apart from the boundary), or one must be completely enclosed in the other. For example, the sequence $1, 2, 3, 4, 1, 2, 3, 4$ has a loop count of only one. Even though four loops exist, because their position all intersect, the loop count for the sequence is only one. This pattern is typical of productions that are in the form A → a b c d A e. It is only one loop, but there are four states in that one loop. To make sure such loops are only counted once, no overlapping loops are allowed to contribute to a loop count.

Figure 5.4 shows an example for the grammar:

```
A → 'a' 'b' 'c' 'd' A 'e'
A → ϵ
```

Enclosed loops

Completely enclosed loops (such as the loop $6, 9, 6$ in $1, 4, 6, 9, 6, 3, 1$) also contribute to the loop count of a stack portion.

Consider the example grammar

```
S → 'z'
S → 'a' 'b' S 'd'
S → 'a' 'c' S 'e'
S → 'a' 'c' 'a' 'c' S 'f'
```

that produces the parsing automata shown in Figure 5.5.

There is a large loop, $1, 4, 6, 3, 1$ and a smaller loop $6, 9, 6$. Each occurrence of the loop $6, 9, 6$ on the stack requires that an 'e' or 'f' be shifted at some point in the future. This means that, if the token sequence 'fff' was

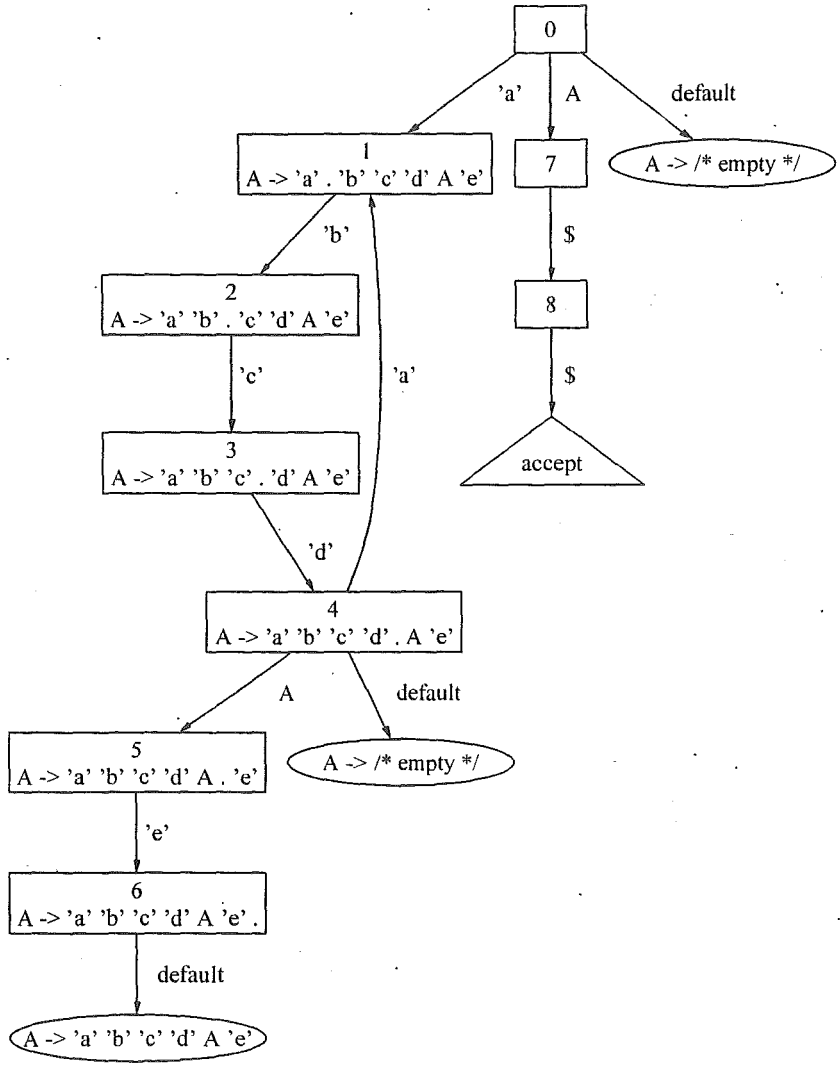

Figure 5.4: Example parser showing loop involving four states: 1, 2, 3, and 4

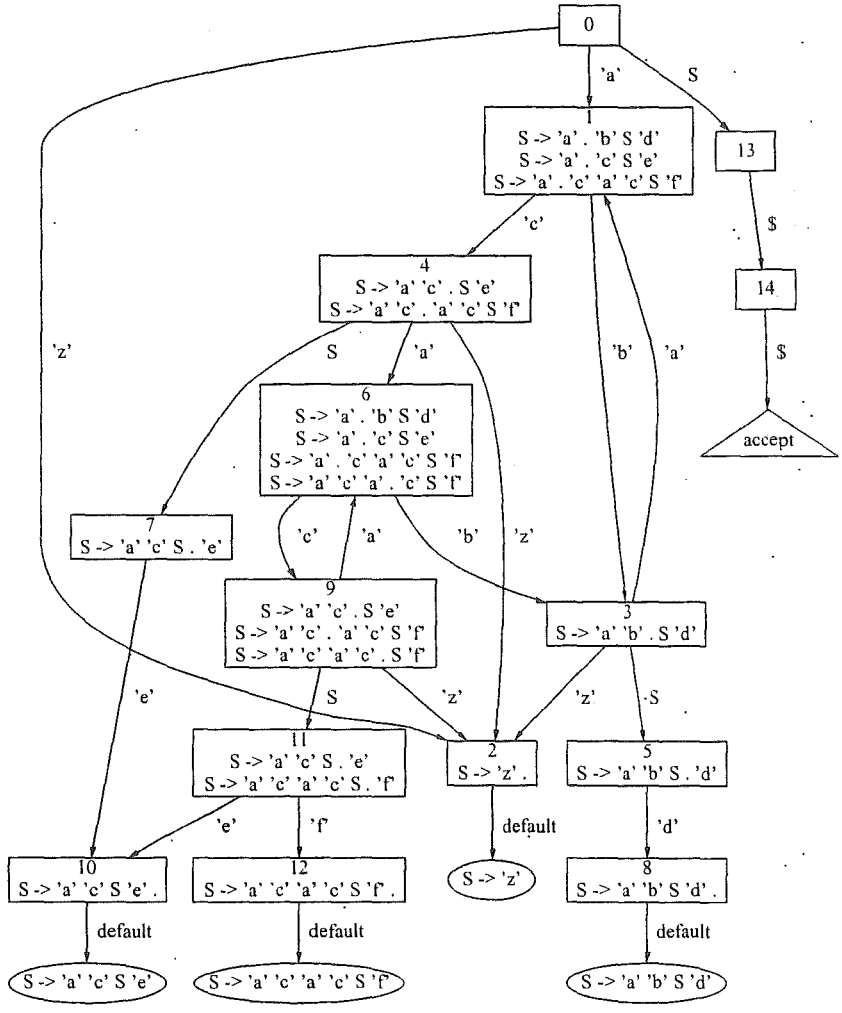

Figure 5.5: Example parser showing nested loops

a prefix of the remaining input, the loop would have to be traversed three times.

5.4 Implementation

All of the simplifying assumptions specified in Section 5.3 were used. Although one of the assumptions does not preserve the least-cost repair property of the search, the conditions under which it arises are unusual enough that it is almost certain that no such construct exists in the grammars used for the experimental results.

Limiting loops imposes no space overhead, but does have the extra time overhead of counting loops.

Chapter VI

The *EquivEdges* algorithm

The *EquivEdges* algorithm, developed as part of the thesis, identifies pairs of edges that produce identical configurations. That is, any repair possible from one edge is also possible from the other edge. The two edges are, from the point of view of the repair algorithm, equivalent, though both are required for normal parsing. One of the edges, therefore, may be safely eliminated.

For example the programming language tokens '+' and '*' would be equivalent in Java— inserting either during a search leads to the same set of continuations. However, for C, the token '*' is used for both multiplication and pointer dereferencing. In certain contexts, '+' and '*' may be equivalent, and in others, they might not.

An explanation of Dain's [26] work in the area is given, followed by an example where equivalence is incorrectly determined by her method. An improved, parser-specific approach is presented that finds more equivalent cases, and does not find incorrect equivalent symbols. The algorithm is described, and applied to a number of programming language grammars. Some limitations are discussed, and runtime overhead is examined.

6.1 Motivation

Dain [26] introduces the idea of two equivalent tokens, and gives the following rather weak condition sufficient for equivalence:

$a \sim b$ for terminal symbols a, b of a CFG $G = (N, \Sigma, P, S)$, if the condition

$$A \to \alpha a \beta \text{ is in } P \text{ if and only if } A \to \alpha b \beta \text{ is in } P$$

holds for all productions in P.

```
['LONG', 'UNSIGNED', 'DOUBLE', 'INT', 'VOID', 'CHAR', 'FLOAT',
'SHORT', 'SIGNED']
['<<=','>>=','&=','|=','%=','+=','-=','*=','/=','^=']
['<<', '>>']
['TYPEDEF', 'STATIC', 'REGISTER', 'EXTERN', 'AUTO']
['CONTINUE', 'BREAK']
['<', '>=', '>', '<=']
['++','--']
['CONST', 'VOLATILE']
['FLOATINGconstant', 'INTEGERconstant', 'CHARACTERconstant',
'OCTALconstant', 'HEXconstant']
['.', '->']
['STRUCT', 'UNION']
['==','!=']
['+', '-']
['~', '!']
['/', '%']
```

Figure 6.1: Equivalence classes for the C language

Dain then goes on to prove that if a, b are terminals of a CFG such that $a \sim b$ then the continuations of ua = continuations of ub for any u in Σ^*.

For two terminal symbols (a, b) that are equivalent, one may be safely discarded when searching for an error. This is because any possible configuration generated from inserting that token will also be generated by inserting the other equivalent token. No potential least cost repairs will be missed by discarding one of the equivalent symbols. If the two symbols do not have identical insert costs, then the symbol with the higher insert cost must be eliminated.

Figure 6.1 shows the equivalence classes for tokens in the C programming language. From the 88 tokens in C, 53 tokens (60%) are spread among 15 equivalence classes.

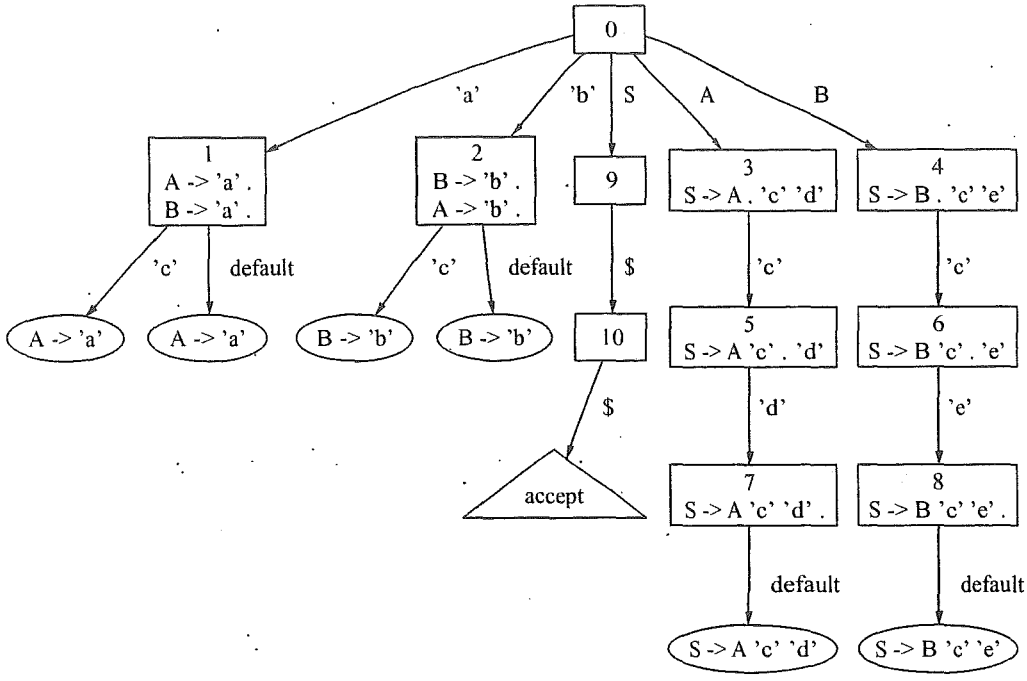

Figure 6.2: Parser generated by *bison* showing incorrectness of Dain's equivalence definition for error correction.

6.2 *Incorrect repair using CFG-based equivalence*

While Dain's definition of equivalence is valid for tokens in a CFG, it can fail when applied to an LR-based parser generated from that CFG. This is because the LR-family of parse tables may only recognise a subset of the language described by the CFG as a result of a shift/reduce or reduce/reduce conflict. The problem arises when two symbols a,b that are equivalent in a grammar are not equivalent in the parser. That is, the set of continuations from each symbol is different.

For example, the parser in Figure 6.2 was generated by the *bison* parser generator. The grammar file used as input is:

```
S → A 'c' 'd'
S → B 'c' 'e'
A → 'a'
B → 'a'
B → 'b'
A → 'b'
```

Note that a $\sim$ b by Dain's definition. During generation, *bison* reports two reduce/reduce conflicts. There is one conflict in state 1 and one conflict in state 2. Both conflicts involve the terminal `'c'`. In state 1, the reduction A $\rightarrow$ a was chosen over B $\rightarrow$ a and in state 2, the reduction B $\rightarrow$ b was chosen over A $\rightarrow$ b. In the absence of any conflict resolution directives, *bison* uses the order of rule definitions to resolve conflicts, thus the unusual order of the rules in this example. The resulting parser can only parse two sentences, `'acd'` and `'bce'`. There are also two sentences that are in the language described by the CFG, but cannot be parsed by the parser due to the resolution of grammar conflicts by *bison*: `'bcd'` and `'ace'`. Note that the grammar is not ambiguous; an LR(2) parser generated from this grammar, using two tokens of lookahead, would have no conflicts because the two states with conflicts in, state 1 and state 2, would each be split into two states.

Now assume that b is chosen from the equivalent set {a,b} to be discarded during a repair, based on a $\sim$ b. If the parser is fed the erroneous input `'ce'` (resulting in an error immediately in state 0), it will not be able to repair the input by inserting the token `'b'`. This will lead to the alternative (and possibly not least-cost) repair of deleting the remaining input and inserting `'acd'`. There is a similar case for the erroneous input `'cd'` and the choice of *a* as the discarded symbol when repairing.

This example shows that using CFG-based equivalence to discard tokens while searching for a repair may lead to incorrect (not least-cost) repairs, regardless of which token is chosen from the equivalence set.

6.3 A Parser-based approach

Dain approached the problem of equivalent symbols from a grammar-based point of view. A more powerful alternative was found by searching for equivalent symbols on a state-by-state basis in the generated parser. The parser-based approach also avoids the problem that may occur in Dain's grammar-based approach where a parser recognises a subset of the grammar due to restrictions in the parsing method.

The parser-based approach calculates, on a state-by-state basis, the possible insert-string/stack pairs for each shift action. If two symbols have the same set of insert-string/stack pairs then the two symbols are equivalent from that state.

Note that the algorithm described here does not find all pairs of equivalent symbols. There are still constructs in the grammar which produce a parser that has equivalent symbols, but they are not found. Finding an algorithm to enumerate all of the equivalent symbols in a parser appears to be difficult. The problem is essentially one of equivalence of deterministic push-down automata (DPDA), a problem which has only recently been shown to be decidable [69, 64].

Consider, for example, two CFGs C_1,C_2, with start symbols S_1 and S_2 respectively, and two DPDAs P_1 and P_2 which recognise the respective languages described by C_1 and C_2. That is, P_1 recognises strings derived from S_1, and P_2 recognises strings derived from S_2.

A new CFG, C_3, with start symbol S_3, can be created by combining C_1 and C_2 and adding the productions $S_3 \rightarrow aS_1$ and $S_3 \rightarrow bS_2$. A DPDA which recognises the language generated by S_3 will have to be able to recognise both a and b from the starting state. After recognising an a, a string derivable from S_1 will have to be recognised, and after b, a string derivable from S_2 will have to be recognised. The DPDA P_1 recognises a string derivable from S_1, and likewise with P_2 and S_2. Therefore, when considering the equivalence of a and b (that is, they have the same set of continuations) from the start state of the DPDA P_3 we need to consider the equivalence of DPDA P_1 and DPDA P_2. Therefore the problem of finding equivalent symbols is the same as determining the equivalence of two deterministic pushdown automata.

6.3.1 *Overview of EquivEdges algorithm*

Essentially, for each edge e from a state S, the goal is to discover every possible continuation from that edge. If two edges from the same state have the same set of possible continuations, then they are equivalent. Equivalence can be determined at parser generation time. For efficiency and complexity reasons, continuation sets are restricted in such a way that some pairs of equivalent symbols will not be found (see Section 6.5), but all pairs of equivalent symbols that are found are indeed equivalent; there are no false positives.

As an example, we will use the grammar:

```
S → 's' 'i' D
S → 's' 'j' D
S → 's' 'k' D
S → 's' 'k' 'x' 'q'
D → 'x' 'z'
D → 'x' 'z' 'p'
D → 'y' 'z'
```

The parser produced by bison for this grammar is shown in Figure 6.3. This grammar and parser will be used in the explanation of the *EquivEdges* algorithm.

6.3.2 *Main data structures for EquivEdges algorithm*

Figure 6.4 shows the main data structure used by the equivalence algorithm. The components of the data-structure are described below.

State The state in the parser from which we are trying to find equivalent edges.

Insert-level The length of the token sequence we are testing for equivalence.

Edge The tokens that are being investigated for equivalence. Each token is a shift (edge) from the state in the parser from which we are trying to

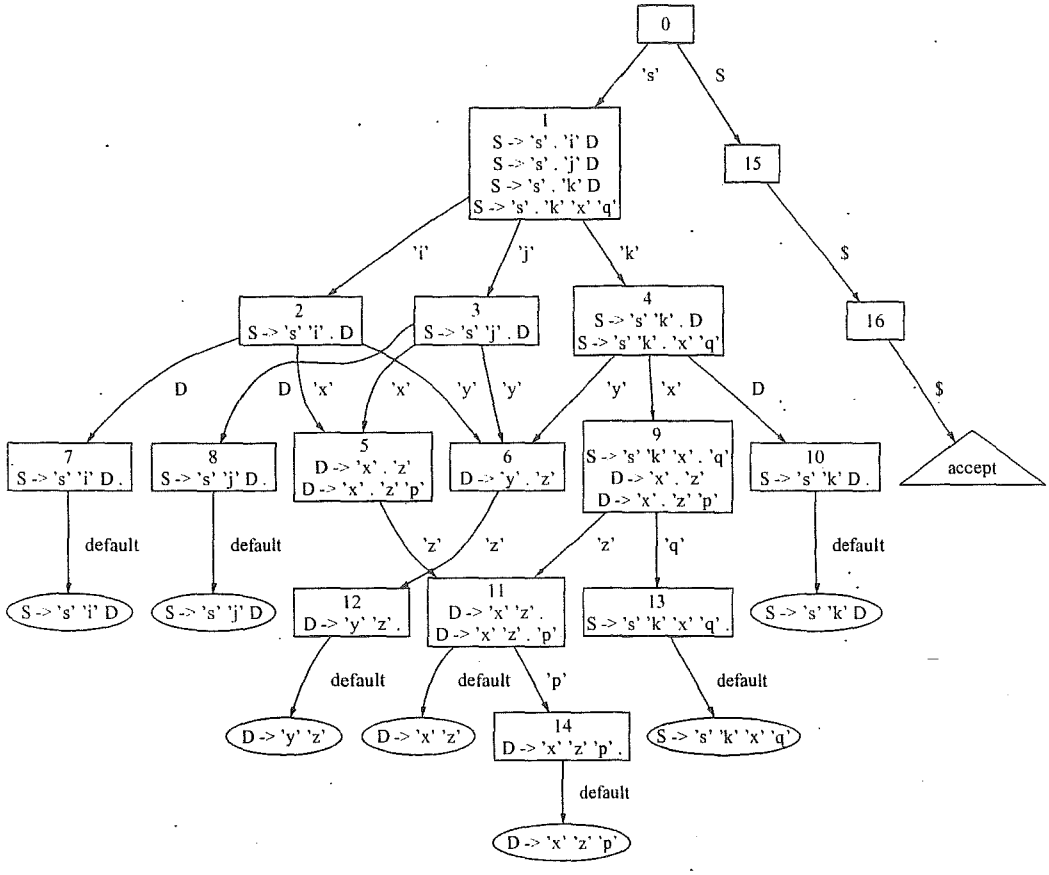

Figure 6.3: Example parser for illustrating the *EquivEdges* algorithm

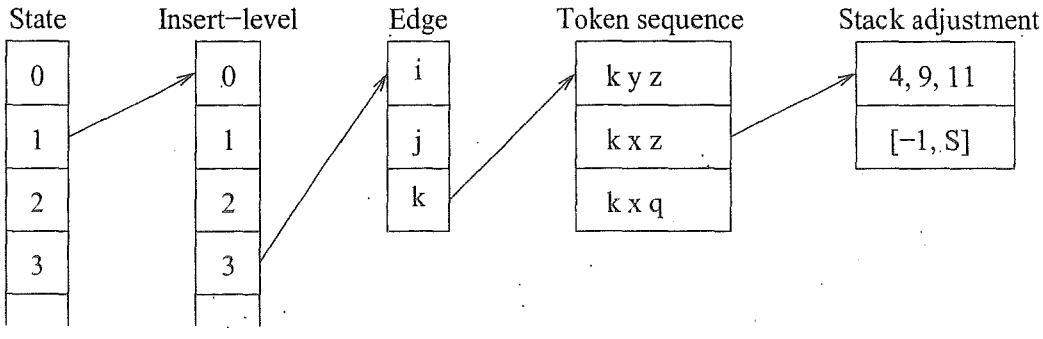

Figure 6.4: An example of the main data structure for the equivalence algorithm.

find equivalent edges. In the example, state 1 has three shifts: i, j, and k.

Token Sequence The possible strings of symbols resulting from a particular shift (and including that shift). The length is the same as the Insert-level. In Figure 6.4, there are three possible token sequences of length 3 from state 1 that start with the token k: `k y z`, `k x z`, and `k x q`.

Stack adjustment The adjustment to the stack resulting from the insertion of the specified token sequence. In the example, from state 1, after inserting the token sequence `k x z`, the stack (with state 1 at the top) could be adjusted in one of two possible ways. The first stack adjustment adds the state sequence 4,9,11 to the stack. The second stack adjustment, denoted by [−1,S], indicates a reduction one state (the −1) back past the starting state (state 1) and from the new top-of-stack (always state 0 in this example), the state resulting from a goto on S (state 15) would be put on the stack. Further possible continuations are not pursued subsequent to a reduction that goes back beyond the state from which we are trying to determine equivalent symbols. (See Section 6.5.3)

6.3.3 Definition of equivalence used in the equivalence algorithm

The following definition of equivalence is used:

> $a \subseteqq b$ $(a, b \in T)$ from a state S at insert-level l if $\forall$ insert-sequences $a, t_2, \ldots, t_l$ $(t_i \in T \cup NT)$ $\exists$ $b, t_2, \ldots, t_l$ such that $t_2, t_3, \ldots, t_l$ are identical for both a and b and the set of stack-adjustments for $a, t_2, \ldots, t_l$ covers the same continuation possibilities as the set of stack-adjustments for $b, t_2, \ldots, t_l$.
>
> $a \equiv b$ if and only if $a \subseteqq b$ and $b \subseteqq a$

Given two token-sequences i_1, i_2 with corresponding stack adjustments of s_1 and s_2 respectively, the algorithm for determining equivalence of the insert-

sequence/stack-adjustment pairs is shown in Listing 6.1. This algorithm must be applied to all pairs for a and b to determine the equivalence of a and b.

Line 2 of the comparison algorithm checks that the two token sequences are the same (the first symbol of each, however, will always be different; it is the equivalence of these two symbols that we are determining). If the two token sequences are identical, then line 4 checks the types of the stack-adjustments.

A stack adjustment is one of two types, either a *sequence* type, where a sequence of states is added to the stack, or a *reduction* type, where a reduction is performed back past the state where equivalence is being tested.

Two reduction-type stack-adjustments have the same set of continuations if the number of states to pop is the same for both, and the goto non-terminal is the same for both (lines 7–13). Two sequence-type stack-adjustments have the same set of continuations if their lengths are the same and the last state is the same. This seems counter-intuitive initially: if two sequences of states have different intermediate states, then a reduction that goes back to an intermediate state will likely have a different result for each sequence. This is never a problem, because reductions to intermediate states are never done. They never need to be done because an insert-sequence can include the goto symbols that would have resulting from doing those reductions.

If the lengths of two sequence-type stack-adjustments are different, or the final state is not the same for both, they are 'maybe-equivalent' ($\stackrel{?}{=}$); we cannot determine equivalence at this insert-level.

6.3.4 Implementation of equivalence definition

The equivalence definition for a and b is realised as a matrix, where rows represent token-sequences of length l beginning with a in conjunction with a stack-adjustment resulting from inserting that token-sequence. Columns similarly represent the possibilities for b. Elements of the matrix correspond to the relation returned by the function defined in Listing 6.1.

For the example parser in Figure 6.4, for symbols i and j from state 1 at insert-level 2, the matrix is shown in Table 6.1.

Elements in the bottom-most row represent the $\exists$ part of the definition

Listing 6.1: The comparison algorithm for two (insert-sequence,stack-adjustments) pairs.

```
1  proc compare_insert/stack(i1,s1,i2,s2):
2      if i1 ≠ i2:        [first symbol of i1 and i2 not included in comparison]
3          return '≠'
4      if type(s1) ≠ type(s2):          [type can be reduction or sequence]
5          return '≠'
6      if type(s1) = ''reduction'':
7          [a reduction-type has a number of states to pop, and a goto-symbol]
8          if states_to_pop(s1) ≠ states_to_pop(s2):
9              return '≠'
10         elif goto_symbol(s1) ≠ goto_symbol(s2):
11             return '≠'
12         else:
13             return '≡'
14     else: [stack-type is sequence-type]
15         if |s1| ≠ |s2|:
16             return '≠'
17         elif last_state(s1) = last_state(s2):
18             return '≡'
19         else:
20             return '≠'
```

	$(jx\ \ 3,5)$	$(jy\ \ 3,6)$	$(jD\ \ -1,S)$	
$(ix\ \ 2,5)$	$\equiv$	$\not\equiv$	$\not\equiv$	$\equiv$
$(iy\ \ 2,6)$	$\not\equiv$	$\equiv$	$\not\equiv$	$\equiv$
$(iD\ \ -1,S)$	$\not\equiv$	$\not\equiv$	$\equiv$	$\equiv$
	$\equiv$	$\equiv$	$\equiv$	

Table 6.1: Comparison matrix for tokens i and j from state 1 at insert-level 2 for the parser in Figure 6.4

a-summary	b-summary	relation
$\equiv$	$\equiv$	$a \equiv b$
$\not\equiv$	$\not\equiv$	$a \not\equiv b$
$\not\cong$	$\not\cong$	$a \not\cong b$
$\equiv$	$\not\equiv$	$a \subseteq b$
$\equiv$	$\not\cong$	$a \subseteq\!\!\!\!\equiv\; b$
$\not\cong$	$\not\equiv$	$a \not\subset b$

Table 6.2: The possible relations between a and b given the relation of the summaries

	$(jx\ \ 3,5)$	$(jy\ \ 3,6)$	$(jD\ \ -1,S)$	
$(kx\ \ 4,9)$	$\not\cong$	$\not\equiv$	$\not\equiv$	$\not\cong$
$(ky\ \ 4,6)$	$\not\equiv$	$\equiv$	$\not\equiv$	$\equiv$
$(kD\ \ -1,S)$	$\not\equiv$	$\not\equiv$	$\equiv$	$\equiv$
	$\not\cong$	$\equiv$	$\equiv$	

Table 6.3: Comparison matrix for tokens j and k from state 1 at inset-level 2 for the parser in Figure 6.4

for a particular token-sequence and are found by choosing the "best" ($\equiv$ better than $\not\cong$ better than $\not\equiv$) value in that column. The entire bottom row is summarised to find the truth of the relation $j \subseteq\!\!\!\!\equiv\; i$. The worst value in the bottom-most row gives the summary.

Combining both summaries gives the final relation between two symbols from a particular state at a particular insert-level. The possible relations are listed in Table 6.2.

The two symbols i and j then, are equivalent from state 1. One may be safely discarded when searching for a repair because all continuations after inserting i are also possible from j, and vice versa.

For symbols j and k from state 1 at insert-level 2, the matrix is shown in Table 6.3.

We can see in this matrix the maybe-equals relation returned by the procedure in Listing 6.1. The final relation between j and k for insert-level 2 at state 1 is $j \not\cong k$. We should try a higher insert-level to find a definite

	$(jxz\ \ 3,5,11)$	$(jxz\ \ -1,S)$	$(jyz\ \ -1,S)$	$(jD\ \ -1,S)$	
$(kxz\ \ 4,9,11)$	$\equiv$	$\not\equiv$	$\not\equiv$	$\not\equiv$	$\equiv$
$(kxz\ \ -1,S)$	$\not\equiv$	$\equiv$	$\not\equiv$	$\not\equiv$	$\equiv$
$(kxq\ \ -1,S)$	$\not\equiv$	$\not\equiv$	$\not\equiv$	$\not\equiv$	$\not\equiv$
$(kyz\ \ -1,S)$	$\not\equiv$	$\not\equiv$	$\equiv$	$\not\equiv$	$\equiv$
$(kD\ \ -1,S)$	$\not\equiv$	$\not\equiv$	$\not\equiv$	$\equiv$	$\equiv$
	$\equiv$	$\equiv$	$\equiv$	$\equiv$	

Table 6.4: Comparison matrix for tokens j and k from state 1 at inset-level 3 for the parser in Figure 6.4

relation.

For insert-level 3 from state 1, testing symbols j and k, the matrix is shown in Table 6.4.

Note that the token-sequence kxz has two possible stack-adjustments. The reduction-type stack adjustment for kxz is $(-1, S)$ and is found by performing the default reduction from state 11 followed by the default reduction in state 10, taking the stack one state prior to state 1, our initial state, and making a goto transition on symbol S.

The final relation for the matrix is $j \subseteqq k$. j may be safely discarded when searching for a repair only if the insert$(j) \geq$ insert(k). k can not be safely discarded based on this information (it may, however, subsequently found to be equivalent to another symbol from the same state, though will not be in this example).

At a given level l, for each pair of edges a, b, a single relationship is derived from a matrix of the relationships between all possible token-sequence/stack-adjustment pairs for each edge: First, each row and column of this matrix is summarised by taking the 'best' relationship from that row/column. An example is seen in the extreme row and column in Table 6.4. Secondly, the row summary and the column summary are further summarised into one relation each by taking the worst of the summary. For Table 6.4, the right-most column (the row summaries) is summarised as $\not\equiv$ (the worst value). The bottom-most row (the column summaries) is summarised as $\equiv$ (the worst, and only, value). Using Table 6.2, these two summaries of summaries give us

the relation $\subseteqq$, that is, $j \subseteqq k$.

6.3.5 Algorithm description for finding equivalent tokens

Listing 6.2 describes the algorithm used to find equivalent tokens. The entry point to the algorithm is the function `find_equiv_edges` on line 82. It first initialises the two main groups of sets, defined on lines 3 and 4.

Next, all the level 1 sets (insert-sequences of length one) are found by calling the function `do_level_1` (line 18) for all states. The function `do_level_1` finds all the insert-sequence/stack-adjustment pairs from a state where the insert-sequence is length one. First, because of ϵ rules, it may be possible to get to other states without inserting any symbols. To find all the length-one insert-sequence/stack-adjustment pairs, it is necessary to find the set of pairs at level zero, as returned by the function `do_level_0` on line 7, then insert one symbol from the state of each pair, then insert as many zero-length pairs as possible to find the final set of length-one insert-sequence/stack-adjustment pairs.

Finally, once all the length-one insert-sequence/stack-adjustment pairs have been found, they are searched for equivalent symbols by the function `find_equiv` on line 39.

The `find_equiv` function returns a set of eliminated tokens given the set of insert-sequence/stack-adjustment pairs. The operation of this function is described informally in Section 6.3.4.

To find equivalent edges at level n ($n > 1$), for state S_i, only $L_{n-1}[S_i]$ and any subset of the sets in L_1 are required (see function `do_level_n` on line 74). Once L_1 has been populated, all levels above n may be done depth-first (per state) or breadth first (per level). The algorithm in Listing 6.2 illustrates a breadth-first (line 87) implementation.

Listing 6.2: The algorithm for finding equivalent edges.

```
1 state S_i
2 insert-level n
3 set of insert-sequence,stack-adjustment pairs L_n[S_i]
4 set of tokens that may be discarded during search E[S_i]
5
6 [return the set of stack-adjustments resulting from inserting nothing from S_i]
```

```
7  proc do_level_0 (S_i):
8      L0 = {S_i}
9      for S where ∃ a goto on a nullable non-terminal from S_i to S:
10         add S to L0
11         L0 = L0 ∪ do_level_0(S)
12     for l,g where ∃ a reduction g → lhs and l = |lhs|:
13         add (l,g) to L0
14     return L0
15
16
17 [populate the set L_1[S_i]]
18 proc do_level_1(S_i):
19     [first find the stack-adjustments resulting from inserting nothing ...]
20     L_0[S_i] = do_level_0(S_i)
21     [... then, from each sequence-type stack-adjustment ...]
22     for j in L_0[S_i]:
23         if type(j) = ''sequence'':
24             [... insert a terminal/non-terminal ...]
25             for S where ∃ a shift or goto on a symbol t from tos(j) to S:
26                 add t,j.S to L_1[S_i]
27     [... then, find resulting stack-adjustments by doing reductions.]
28     while change in L_1[S_i]:
29         for t,a in L_1[S_i]:
30             if type(a) = ''sequence'':
31                 L_0[S_i] = do_level_0(tos(a))
32                 for j in L_0[S_i]:
33                     add t,a.j to L_1[S_i]
34
35     E[S_i] = E[S_i]∪find_equiv(L_1[S_i])
36
37
38 [P is set of insert-sequence,stack-adjustment pairs.]
39 proc find_equiv(P):
40     E = {} [set of eliminated tokens]
41     foreach pair of tokens t_i,t_j starting an insert-sequence in P:
42         I = set of insert-sequence/stack-pairs in P starting with t_i
43         J = set of insert-sequence/stack-pairs in P starting with t_j
44         two-dimensional array M[elements of I][elements of J]
```

```
45          for i,c in I:
46              for j,d in J:
47                  M[i,c][j,d] = compare_insert/stack(i,c,j,d)  [Listing 6.1]
48          for i,c in I:
49              Isumm[i,c] = best(M[i,c][*])  [ordering is: '≡' > '≩' > '≠']
50          for j,d in J:
51              Jsumm[j,d] = best(M[*][j,d])
52          summI = worst(Isumm[*])
53          summJ = worst(Jsumm[*])
54          r = relation(summI,summJ)
55          if r = '⊂':
56              if insert-cost(t_i) ≥ insert-cost(t_j):
57                  add t_i to E
58          elif r = '⊃':
59              if insert-cost(t_i) ≤ insert-cost(t_j):
60                  add t_j to E
61          elif r = '≡':
62              if insert-cost(t_i) ≥ insert-cost(t_j):
63                  add t_i to E
64              else
65                  add t_j to E
66      return E
67
68
69  [return relation defined by Table 6.2]
70  proc relation(r1,r2):
71      return one of ≡,≠,≩,⊂,⊃,⫋,⫌
72
73
74  proc do_level_n(S_i,n):  [for n > 1 ]
75      for i,c in L_{n-1}[S_i]:
76          if type(c) = ''sequence'':
77              for j, b in L_1[tos(c)]:
78                  add i.j,c.b to L_n[S_i]
79      E[S_i] = E[S_i]∪find_equiv(L_n[S_i])
80
81
82  proc find_equiv_edges(maxn):
```

```
83      L_n[S_i] = {} ∀ i,n
84      E[S_i] = {} ∀ i
85
86      do_level_1(S_i)∀ i
87      for n in range(2, maxn):
88          do_level_n(S_i, n)∀ i
```

6.4 *Equivalent edges in different grammars*

To gauge the effectiveness of the algorithm in finding edges to eliminate, forty yacc-compatible grammars were downloaded from the internet. The grammars cover many programming languages (including Ada, Beta, C, Cocoa, D, Dylan, Java, Oberon, Pascal, TTCN-3), as well as grammars for configuration files and other structured information (including the *twm* window manager configuration file format, the configuration file format for *wine*, a windows emulator, XKB keyboard description, and the *debian* project's package specification).

The parsing automata resulting from these grammars ranged from very large (TTCN-3, a recently developed language from ETSI, has 2073 states from 1466 rules) to small (turingol, a high-level language for programming turing machines, has 37 states from 13 rules). In comparison, C has 517 states from 313 rules.

Figure 6.5 shows the number of useful edges remaining at each insert-level for the forty parsers. An insert-level of zero contains all edges in the parser. Parsers with a large number of shifts tend to have a smaller percentage of shifts that are useful.

Figure 6.6 shows the percentage of edges remaining at each insert level for the forty parsers. There is a wide range in the percentage of symbols that are found to be useful, down to around 25-30% for some parsers. There is only one parser (the smallest parser in the set, for the turingol language) where no edges were found to be equivalent.

It would be useful to know in advance what percentage of edges will remain as useful edges. Such knowledge would allow a compiler writer to decide whether implementing the *EquivEdges* algorithm is likely to be worthwhile

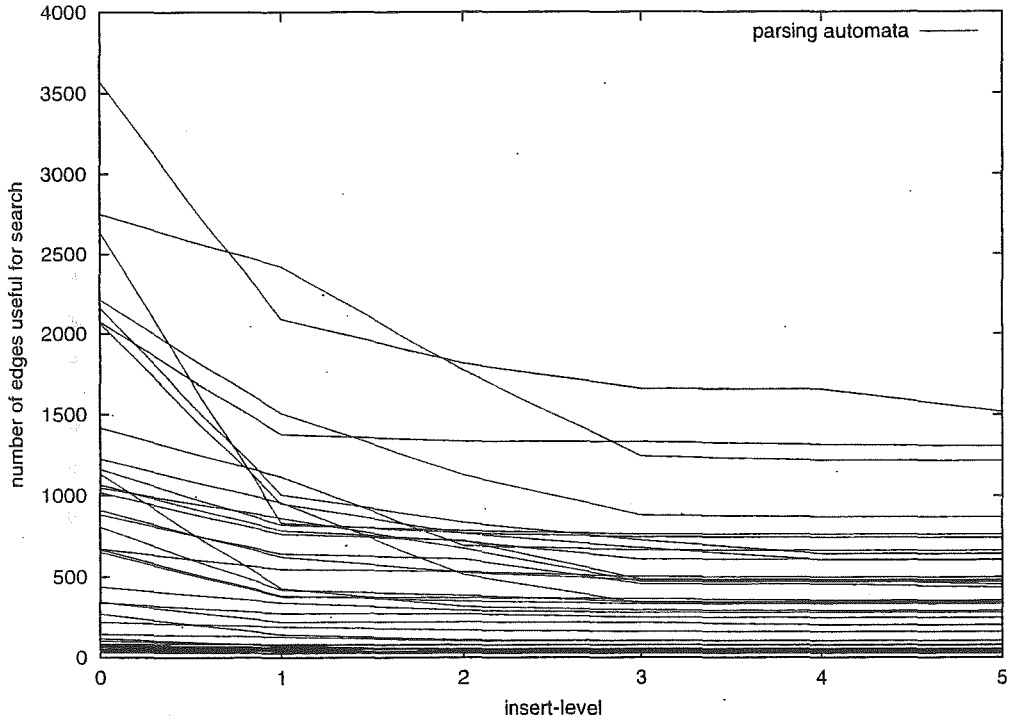

Figure 6.5: Number of useful edges at different insert-levels for forty parsers. An insert-level of 0 contains all edges in the parser.

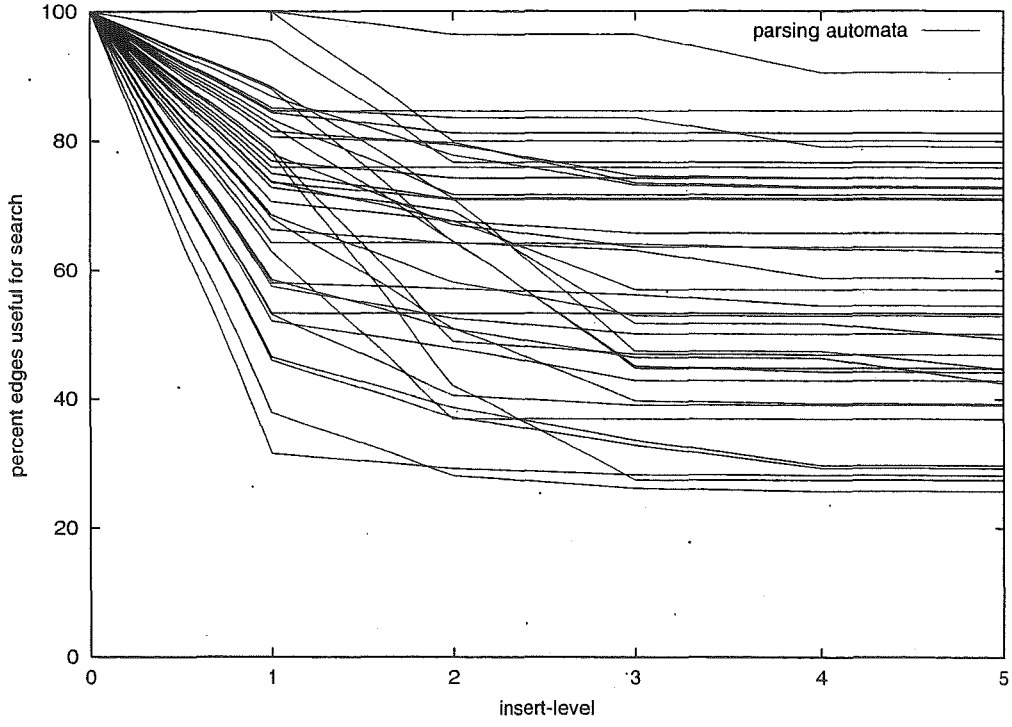

Figure 6.6: Percent of useful edges at different insert-levels for forty parsers.

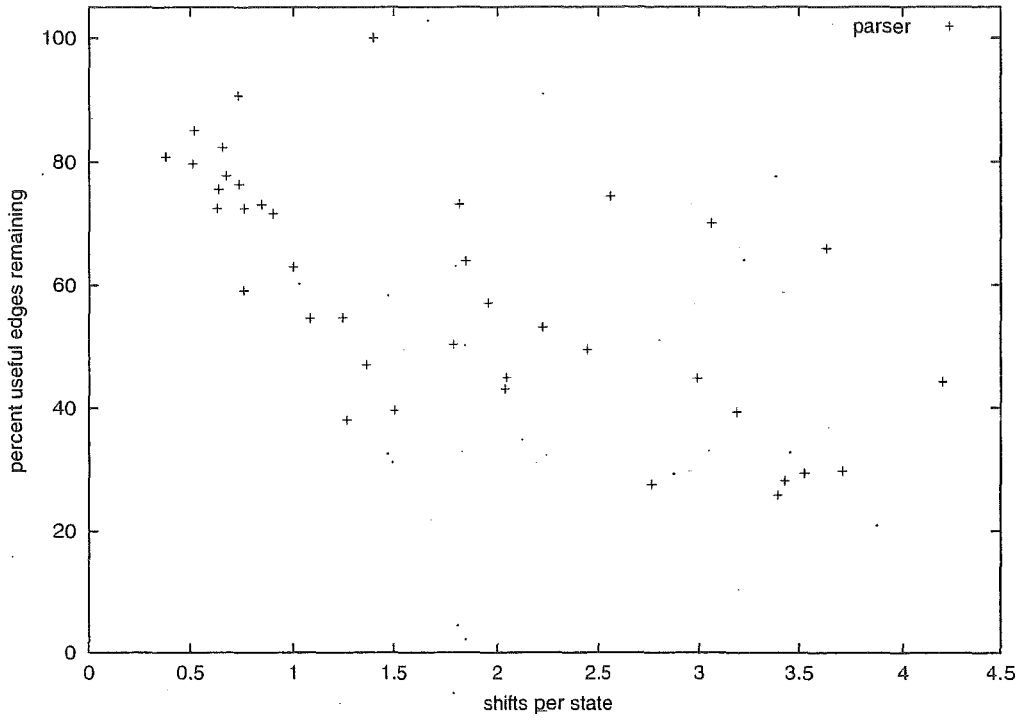

Figure 6.7: Shifts per edge as an indicator of the percent of useful edges (insert level 5).

for a particular parser. An accurate measurement is likely to be very difficult to find without actually implementing the algorithm, but a reasonable heuristic is the number of shifts per state of the parser (the shift-density). A parser with more shifts per state is (quite reasonably) more likely to have shift-edges that are equivalent and can therefore be discarded when searching for a repair.

Figure 6.7 shows that parsers with more shifts per state tend to have a smaller percentage of shift-edges that are useful. The correlation coefficient is -0.66.

6.5 Limitations of the algorithm for finding equivalent edges

There can be some states that have equivalent edges, yet are not found by the algorithm presented in this chapter. This is mainly due to the assumptions

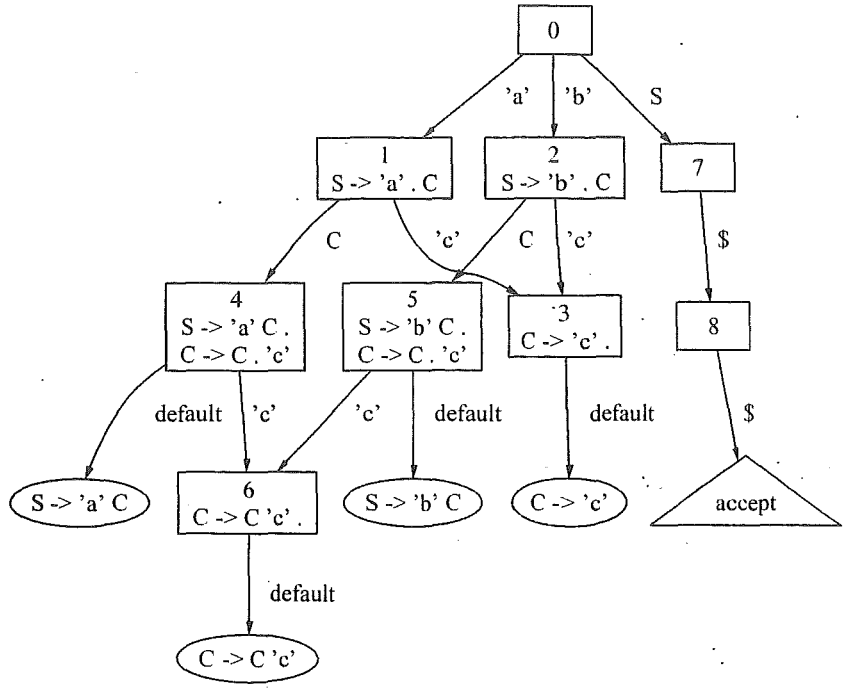

Figure 6.8: Parser with left-recursive constructs that causes problems for the *EquivEdges* algorithm

made in the algorithm for finding equivalent edges.

6.5.1 Left recursion problem

In some cases, an equivalent symbol is not found due to left recursion. For example, the context free grammar:

```
S  →  'a' C
S  →  'b' C
C  →  C 'c'
C  →  'c'
```

has the parser shown in Figure 6.8. The two shifts, `'a'` and `'b'` from state 0 are equivalent; after each shift, one or more `'c'` symbols must occur.

To see why this should cause a problem, consider the possible insert-strings at levels two and three. At insert-level two, the comparison matrix looks like:

	$(bc\ \ 0,2,5)$	$(bC\ \ 0,2,5)$	$(bc\ \ 0,7)$	$(bC\ \ 0,7)$	
$(ac\ \ 0,1,4)$	$\stackrel{\cdot}{=}$	$\neq$	$\stackrel{\cdot}{=}$	$\neq$	$\stackrel{\cdot}{=}$
$(aC\ \ 0,1,4)$	$\neq$	$\stackrel{\cdot}{=}$	$\neq$	$\stackrel{\cdot}{=}$	$\stackrel{\cdot}{=}$
$(ac\ \ 0,7)$	$\stackrel{\cdot}{=}$	$\neq$	$\equiv$	$\neq$	$\equiv$
$(aC\ \ 0,7)$	$\neq$	$\stackrel{\cdot}{=}$	$\neq$	$\equiv$	$\equiv$
	$\stackrel{\cdot}{=}$	$\stackrel{\cdot}{=}$	$\equiv$	$\equiv$	

The summary of this matrix is that $a \stackrel{\cdot}{=} b$. That is, a might be equivalent to b but we do not have sufficient information to tell at this insert-level. For insert-level three, we will only expand out the upper left quarter of the insert-level 2 matrix. The rest of the matrix (involving those entries with the stack $0,7$) can be excluded: a pair of insert-string/stack-adjustments that are equivalent at level n will also be equivalent at level $n+1$. The relevant portion of the insert-level 3 matrix looks like:

	$(bcc\ \ 0,2,5)$	$(bCc\ \ 0,2,5)$	$(bcc\ \ 0,7)$	$(bCc\ \ 0,7)$	
$(acc\ \ 0,1,4)$	$\stackrel{\cdot}{=}$	$\neq$	$\stackrel{\cdot}{=}$	$\neq$	$\stackrel{\cdot}{=}$
$(aCc\ \ 0,1,4)$	$\neq$	$\stackrel{\cdot}{=}$	$\neq$	$\stackrel{\cdot}{=}$	$\stackrel{\cdot}{=}$
$(acc\ \ 0,7)$	$\stackrel{\cdot}{=}$	$\neq$	$\equiv$	$\neq$	$\equiv$
$(aCc\ \ 0,7)$	$\neq$	$\stackrel{\cdot}{=}$	$\neq$	$\equiv$	$\equiv$
	$\stackrel{\cdot}{=}$	$\stackrel{\cdot}{=}$	$\equiv$	$\equiv$	

The insert-level 3 matrix is identical to the insert-level 2 matrix except for the extra token, c, at the end of the insert-strings. The summary of this matrix is also $a \stackrel{\cdot}{=} b$. It is easy to see that, when the relevant portion (the upper left quarter) of this matrix is expanded at insert-level 4, it will again be identical to the insert-level-3 matrix except for another c appended to each insert-string.

This problem results in an infinite recursion when searching for equivalent symbols from state 0. The implementation of this algorithm limits the insert-level depth to cope with this recursion. The infinite recursion also means that a will never be found to be equivalent to b.

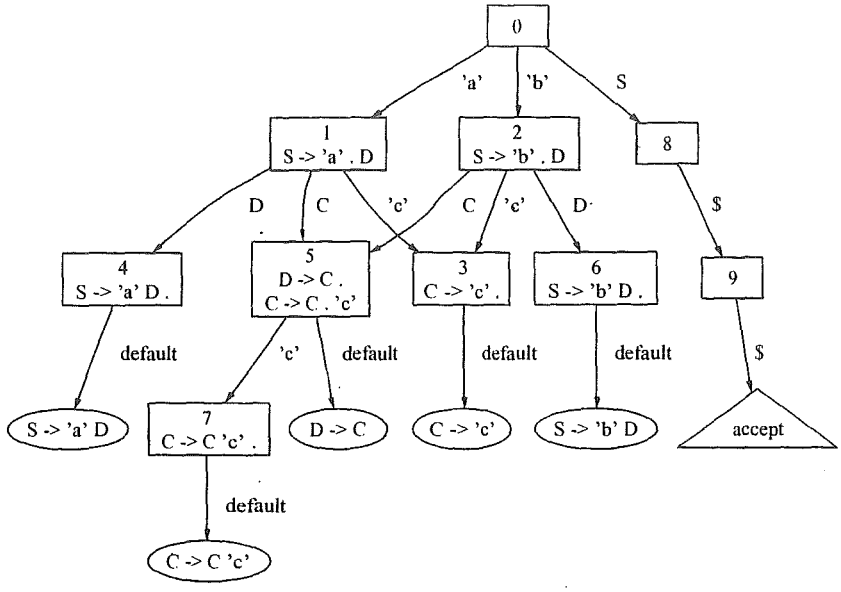

Figure 6.9: Solution to the left-recursion problem by the introduction of a chain rule.

The source of the left-recursion problem is that the same left recursive construct appears in two different contexts: after an a (state 4 of Figure 6.8) and after a b (state 5 of Figure 6.8). A grammar-based solution to the left-recursion problem is to make the left recursion only appear in a single context by introducing a chain rule to start the left-recursion:

```
S → 'a' D
S → 'b' D
D → C
C → C 'c'
C → 'c'
```

The corresponding parser, shown in Figure 6.9, has the left-recursion in only one state, state 5. In this parser, unlike the one shown in Figure 6.8, a will be found equivalent to b.

Changing the left-recursion to right-recursion will also allow a to be found equivalent to b. In practice, the left-recursion problem doesn't occur very often in grammars. None of the grammars used in the experiments in thesis

were modified to remove left-recursion problems although some had left recursion.

6.5.2 Different non-terminal problem

Because the algorithm for finding equivalent edges considers non-terminals as well as terminals, two different non-terminals that derive the same set of strings will be found to be not equivalent. For example, the grammar:

```
S → 'a' A
S → 'b' B
A → 'c'
B → 'c'
```

has the associated parser shown in Figure 6.10. The two edges from state 0, a and b, will not be found to be equivalent, even though they are. At insert-level 2, the insert-string/stack-adjustment pairs (aA $0, 7$) and (bB $0, 7$) are found to be not equivalent because of the different non-terminals.

This limitation is likely to be difficult to fix in general. Grammars intending to be processed by a parser-generator rarely have different non-terminals with identical derivations, as it can easily lead to an ambiguous grammar (where both non-terminals can derive the same sentence).

Grammars that are intended to be read by people may, however, have many different non-terminals with identical derivations to make the grammar easier to understand. For example, the grammar in the TTCN-3 standard [31] has 26 nonterminals directly deriving (and only deriving) the terminal "identifier". The unmodified TTCN-3 grammar has a large number of states with shift/reduce or reduce/reduce conflicts, most of which are caused by different non-terminals with identical derivations. To produce a grammar useful for a LALR(1) parser generator, most of the conflicts will have to be resolved. Doing so will leave a grammar with very few, if any, non-terminals with identical derivations.

6.5.3 Reduction before state problem

When searching for equivalent symbols from a state, reductions that would take the stack back prior to that state (*far reductions*) are not performed.

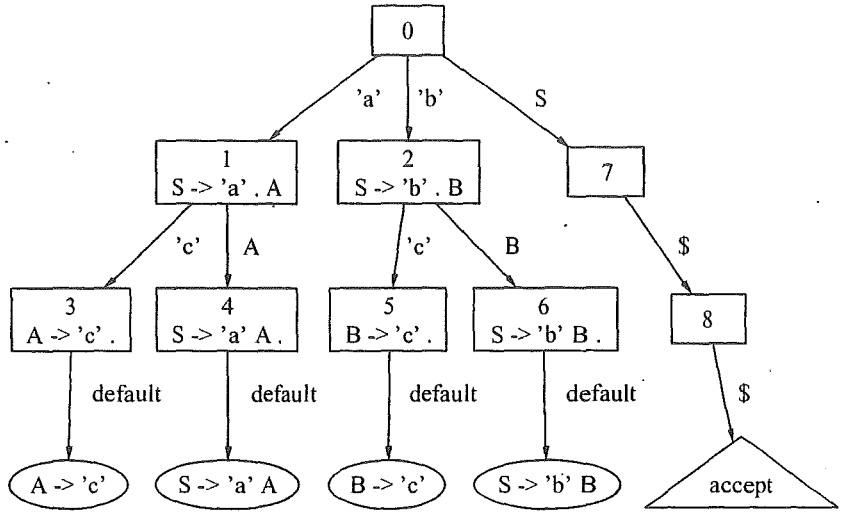

Figure 6.10: Parser showing different non-terminals deriving the same string.

For example, the grammar:

$$
\begin{aligned}
\texttt{S} &\rightarrow \texttt{A} \\
\texttt{A} &\rightarrow \texttt{B} \mid \texttt{C} \\
\texttt{B} &\rightarrow \texttt{'a' 'b'} \\
\texttt{C} &\rightarrow \texttt{'a' 'c'}
\end{aligned}
$$

has the associated parser shown in Figure 6.11. From state 1, the two tokens b and c are equivalent; after inserting either token, the parser will end in state 2, following two reductions. However, because the algorithm for finding equivalent edges does not go back into the stack past the state where searching was started (state 12 in the example), the two tokens a and b will not be found to be equivalent.

At insert-level one, only one insert-string/stack-adjustment pair exists for each token: $(c \quad -2, C)$ and $(b \quad -2, B)$. When testing for equivalence, the two reductions $(-2, C)$ and $(-2, B)$ are said to be not equivalent. More precisely, for the equivalent edges searching algorithm, the relation is "maybe equivalent, but we will never be able to tell". This has the same consequence as "not equivalent".

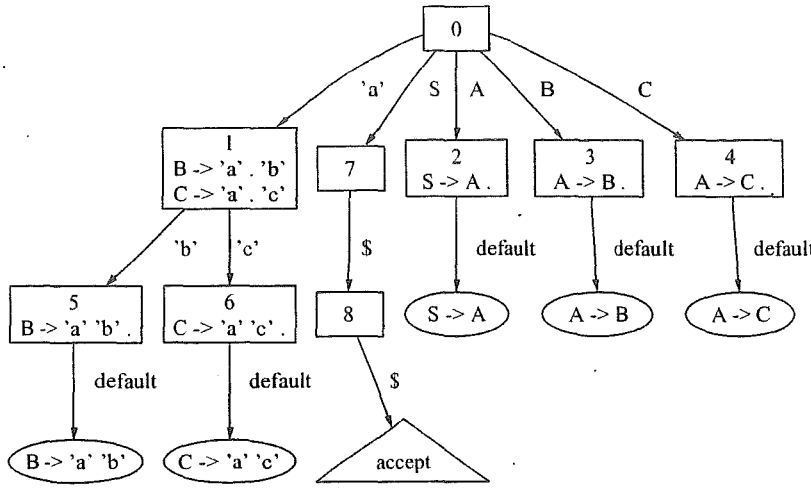

Figure 6.11: Parser illustrating the reduction-before-state problem in state 1.

6.6 *Runtime overhead of EquivEdges algorithm*

Runtime costs are low for the *EquivEdges* algorithm. All edges found to be useless (not useful for a search due to equivalence with another edge) are stored in a table in the generated parser. This table is consulted during a repair for each token from each state.

The algorithm used at runtime for not following useless edges is shown in Listing 6.3. The function `not-useful` performs the table look-up based on the state and edge provided.

Listing 6.3: The *EquivEdges* pruning algorithm

```
1 proc canprune_EquivEdges(config c, edge e):
2     if not-useful(tos(c),e):
3         return true
4     else:
5         return false
```

Chapter VII

Interactions between the three algorithms

In order for the three pruning algorithms to run simultaneously, it must be established that there are no negative interactions between them. That is, it should never be possible that the least cost repair cannot be found. The four combinations that must be checked are:

- *LoopLimit* and *EquivEdges*
- *LoopLimit* and *FolNonTerm*
- *EquivEdges* and *FolNonTerm*
- *LoopLimit* and *EquivEdges* and *FolNonTerm*

7.1 Decision tables for the pruning algorithms

Each of the three algorithms has a restricted domain about which it can make decisions. There are three types of edges, terminals (shift), non-terminals (goto), and reductions (which can be thought of as an edge). For an edge e, an algorithm may make one of three decisions: e is not required (can be pruned), e must be followed (must not be pruned), or e may be followed (that is, we must not prune it, but we cannot say that it must be followed; in effect we "don't know"). For each of the combinations of edge type and decision, the question of whether the decision can be made can be answered by 'Never Made', 'Possibly Made', or 'Always Made'.

The decisions that can be made by the *EquivEdges* pruning algorithm are shown in Table 7.1. For example:

	Reduction-edge	Terminal-edge	Non-terminal-edge
Must be followed	Never Made	Possibly Made	Never Made
Need not be followed	Never Made	Possibly Made	Never Made
Don't know	Always Made	Possibly Made	Always Made

Table 7.1: Decision table showing the edge types that *EquivEdges* is able to make a decision about.

	Reduction-edge	Terminal-edge	Non-terminal-edge
Must be followed	Never Made	Never Made	Never Made
Need not be followed	Possibly Made	Possibly Made	Possibly Made
Don't know	Possibly Made	Possibly Made	Possibly Made

Table 7.2: Decision table showing the edge types that *LoopLimit* is able to make a decision about.

	Reduction-edge	Terminal-edge	Non-terminal-edge
Must be followed	Never Made	Never Made	Always Made
Need not be followed	Possibly Made	Never Made	Never Made
Don't know	Possibly Made	Always Made	Never Made

Table 7.3: Decision table showing the edge types that *FolNonTerm* is able to make a decision about.

- A decision is 'Never Made' about whether a 'Reduction-edge' 'Must be followed'.
- A decision is 'Possibly Made' that a 'Terminal-edge' 'Need not be followed'.
- A decision is 'Always Made' that it 'Does not know' about a 'Reduction-edge'.

The only chance for two algorithms to have a destructive interference when combined is if one algorithm can possibly require an edge-type ('Must be followed') and the other algorithm can possibly prune that edge-type.

From Table 7.4, a summary of the decisions that a pruning algorithm may

	Reduction-edge	Terminal-edge	Non-terminal-edge
EquivEdges	—	required/pruned	—
LoopLimit	pruned	pruned	pruned
FolNonTerm	pruned	—	required

Table 7.4: Possible decisions by pruning algorithms on edges, summarised from Tables 7.1,7.2, and 7.3

make about an edge, we can see that the only two possible destructive interference can arise between *EquivEdges* and *LoopLimit*, where *EquivEdges* requires a terminal edge that *LoopLimit* prunes, and between *FolNonTerm* and *LoopLimit*, where *LoopLimit* prunes a non-terminal edge. As we shall see below, these situations will not lead to problems, because the *LoopLimit* algorithm only prunes an edge once it has already been seen v (the validation length) times, and is a 'stronger' prune, overriding decisions by other pruning algorithms regarding that edge.

7.2 EquivEdges and LoopLimit interaction

Because it is possible that *EquivEdges* may require a terminal-edge to be followed, and *LoopLimit* may prune a terminal edge, we need to be sure that the two algorithms do not interact destructively.

Consider the following grammar:

A → x r A t
A → x r c
A → y r A t
A → y r c

and the corresponding parser, shown in Figure 7.1. It should be clear from the grammar that tokens 'x' and 'y' have the same continuations, and so are equivalent wherever they appear (shifts from states 0, 3, and 4) in the parser. Taking state 3 as an example (state 4 would work equally well), we know the edges 'x' and 'y' are equivalent. If the *EquivEdges* algorithm chooses 'y' as the token to be eliminated (the dashed edge in the figure), the shift on 'x' (the bold edge in the figure) must not be eliminated in order to conserve

the least-cost repair property. The *LoopLimit* algorithm will prune out the shift on 'x' from state 3 if the sequence of inserted symbols for the current configuration has three loops, and the shift on 'x' from state 3 would create a fourth loop (assuming a validation length of 3). One condition satisfying this would be an error in state 1, and a configuration which has inserted the tokens `rxrxrxr`, leading to a stack (from the error state) of 1, 3, 1, 3, 1, 3, 1, 3, containing three loops. The final shift on 'x' from state 3 would push state 1 onto the stack, producing four loops, and so never resulting in a least-cost repair for a validation length of three, and the edge is therefore pruned. Because the *LoopLimit* algorithm determined that the shift on 'x' from state 3, in this context, will never result in a least-cost repair, a shift on 'y' from state 3, in this context, will also never result in a least-cost repair (the two edges are equivalent). The fact that the *EquivEdges* algorithm determined that 'y' could be eliminated on the condition that 'x' not be eliminated is overruled by the *LoopLimit* algorithm determining that 'x' (and, as a result of equivalence, 'y') can, in certain contexts, be eliminated.

Generalising the example, we have (at least) three derivations of a non-terminal:

$$\begin{aligned} A &\rightarrow^* a\alpha A\beta \\ A &\rightarrow^* b\alpha A\beta \\ A &\rightarrow^* \gamma \end{aligned}$$

where $a \equiv b$ in some state s of the parser, and γ is the least-cost insert string for A. The recursion in the derivations of A (Section 5.2.1) generates a loop in the parser, and one of those states in the loop will be state s. If we assume that b is chosen from the equivalent set of $\{a, b\}$ as the symbol to be eliminated, then a must be followed from state s during a search for a repair. If, during a repair, the remaining input is $\beta_1\beta_2 \ldots \beta_v$, where v is the validation length, then $a\alpha$ must be inserted v times for the repair. The edge is only pruned as a result of the *LoopLimit* on the next $(v + 1)$ insertion, when a least-cost repair is no longer possible (Section 5.2.4). If inserting a can no longer result in a least-cost repair, then neither will inserting b, as the two symbols both lead to the same set of continuations. Thus there is no conflict between the *EquivEdges* and the *LoopLimit* pruning algorithms.

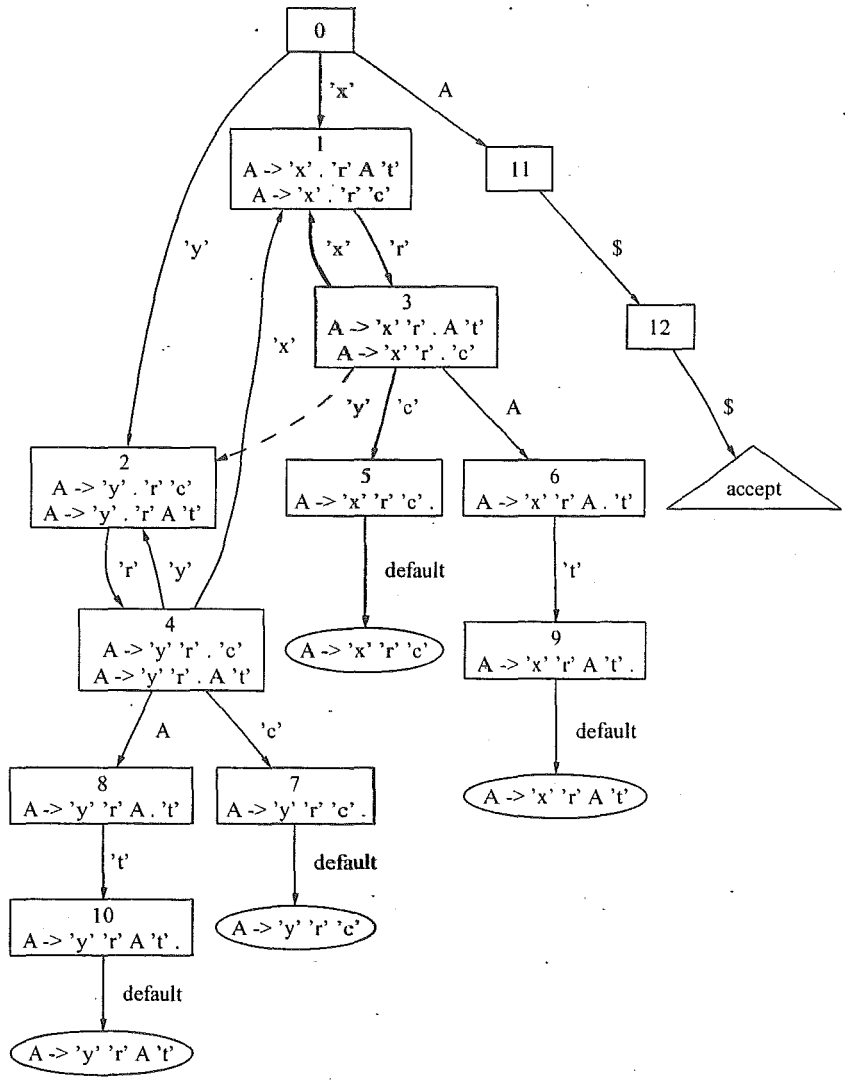

Figure 7.1: Parser showing potential conflict between *EquivEdges* and *LoopLimit*

7.3 *FolNonTerm and LoopLimit interaction*

The *FolNonTerm* algorithm inserts non-terminals, in order to prune reductions which would require a goto on that non-terminal in the future. The *LoopLimit* algorithm may prune the non-terminal edge, thereby denying any repairs that would have come from following that edge.

The interaction between *FolNonTerm* and *LoopLimit* is similar to the interaction between *EquivEdges* and *LoopLimit*. In the latter, we have two equivalent terminal symbols, where one must be followed in order for the other to be pruned, yet because they are equivalent, when inserting one would no longer lead to a least-cost repair, then nor would the other. In the former case, we have instead a goto on a non-terminal edge A, which is equivalent to the insertion of a least-cost derivation of A, followed by a reduction[1]. In the same way, if the *LoopLimit* algorithm determines that inserting a non-terminal A (a goto-edge) would lead to a stack where a least-cost repair would not be found, then inserting the least-cost derivation of A followed by the appropriate reduction(s) would also lead to the same result. Thus the pruning effect of the *LoopLimit* pruning algorithm has no effect on the ability to find a least-cost repair when combined with the *FolNonTerm* pruning algorithm.

[1] Or, possibly, a sequence of reductions

Chapter VIII

Experiment Description

In this chapter, a large collection of Java programs is described which form the basis of our error repair experiments. Some interesting questions are presented, covering the pruning method, grammar, and assignment of costs, which the experiment is designed to answer.

8.1 Previous experiments

While there have been many algorithms in the area of error recovery, there is rarely any experimental evidence provided using real programs. The notable exception is a handful of papers [5, 7, 17, 62, 66, 68] using the now dated Pascal suite of programs analysed by Ripley and Druseikis [63]. This suite comprises 126 Pascal programs written by graduate students, with a total of 414 errors. Of these errors, 88% were single-token errors (requiring a single insert, delete, or replace).

One advantage of having such a small number of programs was that each program could be analysed by hand to decide on what (and where) the syntax errors where, and what the best repair should be, although it was noted that often there was ambiguity about exactly what was the most appropriate repair.

The advantage of a large collection is the broader range of errors that are represented. For a problem where only about 10% of the errors are non-trivial (requiring more than a single insert, delete, or replace), a large collection is essential to get a good idea of the performance of an algorithm.

The two collections have different objectives. The Ripley and Druseikis collection is focused on the *quality* of error repairs, using criteria such as "exactly what a competent programmer might have done". In contrast, the

Java collection presented here, almost 500 times the size of the Ripley and Druseikis collection, was collected in order to compare the *efficiency* of variants of locally least-cost repair. The quality of locally least-cost repairs is already known to be good on the Ripley and Druseikis collection.

8.2 The program collection

Java programs were collected during the period 1998–2000 from two first-year courses at the University of Canterbury, New Zealand. One course was an introduction to programming using Java, the other was an introduction to computer science using Java (following the previous course). Students were given the opportunity at the beginning of the year to opt out of program collection, although typically 95% of students agreed to have their broken programs collected. A wrapper script was placed around the Java compiler that saved a copy of any file that it could not successfully compile. In total, 196,860 'programs' were collected that would not compile. The total number of bytes in the collection[1] is 377,646,704. The mean program size is 1918 bytes, with a standard deviation of 4474 bytes. The mean number of tokens per program is 373.

For privacy reasons, comments in the programs such as `// This is fun` or `/* increment foo */` were replaced with `// comment` and `/* comment */`. In the same manner, strings were replaced with `"string"`. Identifiers were not changed. No record was made of which student submitted which file.

To test for syntax errors (not all programs that fail to compile have syntax errors), two different Java grammars obtained from the internet were used, one from Erik Bäckerud (Appendix A) and one from Dmitri Bronnikov (Appendix B). These two grammars both recognise a slightly different superset of Java (Section 8.5). The Bronnikov grammar was modified to separate the `OP_DIM` token into separate `'['` and `']'` tokens to ensure that both grammars used the same set of legal tokens. The tokens for unimplemented keywords in Java (such as `goto` and `generic`) were also ignored (they do not appear

[1] The collection compressed with the program `bzip2` shrinks to 9,331,213 bytes, a reduction in size by factor of 40, and an average size per compressed program of just under 48 bytes!

Both grammars agree on error token	59,643	30%
Both grammars disagree on error token	28,791	15%
Only Bäckerud grammar detects error	4,281	2%
Only Bronnikov grammar detects error	32,418	16%
Both grammars detect no syntax error	71,757	36%
Total	196,860	100%

Table 8.1: Error detection categories for the collected Java programs

in the grammar, but are reserved words).

Table 8.1 shows the number of programs where there was a difference (or not) in the parsing of a program for each grammar. In order to be able to compare the error-repair qualities of each grammar, the 59,643 programs where both grammars agreed on the error-token were chosen. Even though this eliminates almost half of the programs containing a syntax error, the collection still contains a very wide variety of errors.

8.3 Experiment questions

There are a number of interesting questions that can be answered in part by altering the experimental factors. The main relevant experimental factors here are

- Choice of locally least-cost pruning method.
- Choice of Grammar.
- Choice of Insert/Delete costs.

Secondary variables include the validation length and language. The validation length was set at three— long enough to ensure good repairs, and short enough that a second error is unlikely to be within the validation length (although for a counter-example, see the explanation of the 'Method arguments incorrect' entry in Table 8.2). The language (Java) was fixed.

Each major factor is discussed in the following sections.

```
ProgramFile
        : PackageStatement ImportStatements TypeDeclarations
        | PackageStatement ImportStatements
        | PackageStatement                  TypeDeclarations
        |                  ImportStatements TypeDeclarations
        | PackageStatement
        |                  ImportStatements
        |                                   TypeDeclarations
        ;
```

Figure 8.1: The definition of the `ProgramFile` non-terminal from the Bronnikov grammar.

8.4 *Pruning method*

Here we have the choice of McKenzie's algorithm (the *Default* mechanism with no pruning, described in Section 3.1), with the addition of the pruning methods *FolNonTerm* (Chapter 4), *EquivEdges* (Chapter 6), and *LoopLimit* (Chapter 5), as well as any combination of the latter three. There are eight combinations altogether. The *EquivEdges* algorithm also has a variable associated with it— how deep to look from each state to decide equivalence of symbols. This depth limit was chosen to be 5. See Section 6.4 for more details.

By altering the pruning algorithm we can help answer the question "Which pruning algorithm has the greatest effect at minimising repair costs?" A more complete answer to this question will also quite possibly depend on the choice of grammar and choice of insert/delete costs for the tokens.

8.5 *Grammar*

There are two Java grammars used for testing. They both have a number of different constructs. Selecting each grammar can give some indication of whether different ways of representing the same part of a language can have an effect on error repair.

The Bronnikov (Appendix B) grammar has no ϵ productions. Optional

language constructs tend to be expanded into as many rules as necessary to describe different possible combinations. In contrast, the Bäckerud grammar (Appendix A) grammar makes extensive use of ϵ productions. The Java construct in the Bronnikov grammar shown in Figure 8.1 is represented in the Bäckerud grammar as:

```
compilationUnit : optPackage importList typeDeclarationList ;
```

All three non-terminals in the `compilationUnit` rule are nullable. (Strictly speaking, `ProgramFile` and `compilationUnit` are not the same, because `ProgramFile` is not nullable. If we assume an augmented grammar with production `Z : ProgramFile <eof> | <eof>`, where `<eof>` denotes end-of-input, we get the same behaviour as `compilationUnit`).

The Bronnikov grammar is more hierarchical. The description of a statement, for example, is expressed as:

```
Statement : EmptyStatement
          | LabelStatement
          | ExpressionStatement ';'
          | SelectionStatement
          | IterationStatement
          | JumpStatement
          | GuardingStatement
          | Block
          ;
```

where many of the nonterminals on the right-hand side expanding out to the actual program statements appropriate to that non-terminal.

This contrasts with the Bäckerud grammar which tends to be much 'flatter', as shown in Figure 8.2

8.6 Insert/Delete costs

The assignment of insert and delete costs to tokens has an effect on the quality of a repair and the time taken to find a repair. Four methods have been used to assign insert/delete costs. These methods are explained in the

```
statement : ';'
    | compoundStatement
    | expressionStatement ';'
    | variableDeclaration
    | IF '(' expression ')' statement     %prec LOWER_THAN_ELSE
    | IF '(' expression ')' statement ELSE statement
    | WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | BREAK ';'
    | BREAK simpleSymbol ';'
    | CONTINUE ';'
    | CONTINUE simpleSymbol ';'
    | RETURN ';'
    | RETURN expression ';'
    | FOR '(' forInit optExpression ';' forIncr ')' statement
    | THROW expression ';'
    | SYNCHRONIZED '(' expression ')' statement
    | simpleSymbol ':' statement    %prec NOT_AN_OPERATOR
    | TRY compoundStatement catchList optFinally
    | TRY compoundStatement finally
    | SWITCH '(' expression ')' compoundStatement
    | CASE expression ':' statement
    | DEFAULT ':' statement
    ;
```

Figure 8.2: The definition of the `statement` non-terminal from the Bäckerud grammar.

following sections. Most of the literature that discusses least-cost repairs has little or no discussion on how costs were chosen.

8.6.1 Length-based costs

Used by McKenzie et. al [59] and elsewhere. The cost of each symbol is based on the length of the symbol.

Costs were assigned to tokens equal to the length of that token. The costs ranged from 12 (for the `synchronized` keyword) to 1 (for 24 short tokens, such as '+' and '*'). Literals and identifiers were assigned a cost of 5. In contrast to the Bronnikov grammar, which has one token for all literals, the Bäckerud grammar separates literals into different tokens: strings, ints, doubles, longs, character, true, and false. Even though the length (or approximate average length) of these is different, they were all assigned a cost of 5 to be consistent with the Bronnikov grammar. The delete cost for a token is the same as the insert cost.

8.6.2 Frequency-based costs

The cost of each symbol is based on the frequency with which that symbol appears in programs. There are some difficulties with this approach:

- A large collection of programs is required that can be used to extract the frequencies of symbols.
- Any costs will naturally reflect bias in the collection of the programs from which frequencies are extracted.
- What function should be used to transform a frequency-count of tokens into insert/delete costs for those tokens?

The collection of programs available (nearly 200,000) is certainly large enough to counter the first problem above, although a compiler writer might not necessarily have access to a large collection.

There is undoubtedly some bias of the collection toward novice programmers. For example, there were a few tokens that were not used at all in the

entire collection that an advanced programmer is more likely to use, such as shift-right-equals (>>=). The bias could be considered an advantage, however, as novice programmers are more likely to benefit from useful compiler error messages.

Costs were assigned based on the entropy (information content) in bits [13] calculated for each token based on its frequency in the entire collection of broken Java programs.

For a token with probability p_i, the information content E_i in bits is given by:

$$E_i = -log_2 p_i \tag{8.1}$$

There were a total of 73,422,784 tokens in the programs collected. The most common token was identifier, with 21,190,884 (29%) occurrences, and an insert/delete cost of 2 (see Appendix C for the details). Three tokens did not appear at all, shift-right-equals (>>=), `transient`, and `volatile`. These tokens were assumed to have appeared once to avoid them having infinite information content (resulting in a token with infinite insert cost). This is an acceptable solution to the zero-frequency problem [13] in this context, and resulted in an insert/delete cost of 26 for these tokens. Like length-based costs, delete costs are equal to inserts costs.

8.6.3 Identical costs

The insert and delete costs for all tokens were assigned to the same value. This was used as a control.

8.6.4 Ad-hoc approach

This is a common, but unscientific method. Each symbol is given a cost based on some insight (real or perceived) from the compiler writer. Fischer and Mauney [33] state that "deriving a good set of costs appears to be more of a knack than a mathematical science". This section presents a method of deriving costs that, while certainly far from a mathematical science, is, slightly better than just a 'knack'.

The ad-hoc costs for this experiment were based on the unrepairable

String delimiters missing/incorrect	474	54%
Commenting error	307	35%
String concatenation operator missing	20	2.3%
Method arguments incorrect	17	1.9%
Not a program file	12	1.4%
Truncated method	12	1.4%
Wrong language	7	0.8%
Missing open/close brace	6	0.7%
Miscellaneous	25	2.8%

Table 8.2: Error categories for the most difficult repairs

errors using the *EquivEdges* and *FolNonTerm* algorithms with uniform insert/delete costs using the Bäckerud grammar (see Table 9.1 on page 95).

There were 880 files submitted to the compiler that had errors that were not repairable using this method (although that is not to say that some of these were not repairable by other methods. They merely represented a good selection of programs with difficult-to-repair errors.)

These files were looked at by hand to determine the common causes of difficult to repair errors.

The type of errors that are the most difficult to repair are shown in Table 8.2. Only the first error in each program was categorised. The explanation of the categories is as follows:

- **String delimiters missing/incorrect** Double quotes were sometimes used as a string separator, or unescaped. For example `"foo"bar"baz"` instead of the more likely `"foo\"bar\"baz"` or `"foo"+"bar"+"baz"`. This error was often at the end of a truncated method at the end of a file. Because the string error occurred before the end of the truncated method, it was the first error and did not contribute to the 'Truncated method' entry in the table.

- **Commenting error** Comment opening characters were often transposed (*/ instead of /*), or a single line comment (a line beginning with //) was followed by more than one line of commenting.

```
  public static void printCol(int height, char ch) {
    // comment
    For Outer:=1 TO 3 DO
    IO.stringOut("string")
END
  }
```

Figure 8.3: A method from one of the programs in the collection showing an example of the 'wrong language' error.

- **String concatenation operator missing** Typically a method argument with a number of strings separated by nothing.

- **Method arguments incorrect** Sometimes types of arguments were included in a method invocation. For example, the method invocation `foo(int a, int b, String c)`. Because the three errors in the example are closer than the validation length of 3, any attempt at repair inside the parentheses will include one of those errors and parsing will not be able to continue. A validation length of two would result in three correct repairs (removal of the type names).

- **Not a program file** Some files were lists of numbers or otherwise obviously not a program file.[2]

- **Truncated method** Method definitions were partly completed, often at the end of a file.

- **Wrong language** Fortran or Pascal-like pseudo-code inside a method. Figure 8.3 shows an actual example.

- **Missing open/close brace** Most of these were successfully repaired; in the entire collection of Java programs, the opening brace symbol

[2] Strictly speaking, of course, *all* the files tested were not program files; if they were, they would not have been rejected by the compiler and would not be in the collection. While most files are *close* to being a program, the files in the "Not a program file" category are singled out for their total non-resemblance to any known programming language.

('{') appears approximately 5000 times (0.2%) more than the closing brace symbol[3] ('}')(Appendix C).

- **Miscellaneous** Various unusual errors, including what appear to be cut and paste mistakes.

It was clear that many of the difficult to repair errors stemmed from a lack of Java knowledge by beginner programmers. It is also interesting to note that comment errors and most string errors are lexical errors, not syntax errors. It is surprising that there are so many occurrences of these problems given that the students were using a syntax-highlighting editor, which clearly shows (by using different colours of text) what parts of a program are comments and strings. The use of two characters for comment delimiters appears to have tripped up a number of programmers. Furthermore, the two characters are valid syntactic tokens for what is a purely lexical construct. Having single character comment delimiters that are not tokens in the grammar would almost entirely eliminate the comment errors from the 'difficult-error' collection.

Assignment of ad-hoc costs

All costs were initially assigned an insert and delete cost of 16.

The two leading causes of errors account for about 90% of the difficult repairs. In nearly all those cases, parts of the source file that were intended to be inside a comment or a string were instead read by the parser and interpreted as tokens. To effectively correct such errors, identifiers were given a low relative delete cost. Literals were also given a low relative delete cost, and the comment delimiters / and * were given a low relative delete cost. To encourage closing of blocks and constructs surrounded by parentheses, the closing brace } and closing parenthesis) were both given low relative insert costs. Finally, the string concatenation operator (+) was given a low relative insert cost.

[3] For the two other pairs of delimiters, () and [], the opening delimiter appears more often (0.1% for '(' and 0.01% for '[') than the closing delimiter.

Costs could have been further refined. For example, a few of the incorrect comments looked something like:

```
*/==========================================*/
```

Here the parser sees one times-symbol token, one slash token, a few dozen equal-sign tokens followed by another times-symbol token and a slash token. Reducing the delete cost of an equal-sign would have enabled a rapid repair for this error. This was not done because the costs would become too specialised for the dataset.

Chapter IX

Experimental Results

This chapter presents the results of experiments based on the factors described in the preceeding chapter. There is a section looking at each of the three factors: choice of insert/delete costs, choice of grammar, and choice of pruning algorithm.

There are eight possible choices of pruning algorithm, four choices of cost assignments, and two choices of grammar; 64 combinations altogether. Comparing each pair of combinations, therefore, requires 2016[1] comparisons.

To reduce the number of comparisons required, yet still find the most interesting results, the three factors are considered separately in the following order: cost, pruning algorithm, grammar. All combinations are considered for finding the best cost assignment, then the best cost assignment, which we will see to be ad-hoc costs, is the main cost assignment used for finding the best pruning algorithm combination. Finally, the best pruning algorithm combination in conjunction with the best cost-assignment is used to compare the two Java grammars.

9.1 Results for different insert/delete cost choices

Table 9.1 and 9.2 show the number of errors that were unable to be repaired[2], for all combinations of cost, pruning algorithm combination, and grammar.

The tables show that the ad-hoc cost assignment has fewer unrepairable errors than the other three cost assignments across all pruning methods, and

[1] $2016 = (64 * 63)/2$ A combination a need not be compared with itself, and a compared with b is the same as b compared with a.

[2] That is, 1,000,000 configurations were queued during the search for a repair, but no repair was found.

Bäckerud grammar				
Pruning Algorithm	Cost mechanism			
	Frequency	Length	Uniform	Ad-hoc
Default	1366	4048	2601	537
EquivEdges	1128	1619	1027	131
FolNonTerm	1182	4133	1459	215
LoopLimit	1366	4048	2601	537
EquivEdges FolNonTerm	1064	1368	880	85
EquivEdges LoopLimit	1128	1583	1027	131
FolNonTerm LoopLimit	1182	4133	1459	214
EquivEdges FolNonTerm LoopLimit	1056	1306	880	85

Table 9.1: Comparison of unrepairable errors for the Bäckerud grammar.

Bronnikov grammar				
Pruning Algorithm	Cost mechanism			
	Frequency	Length	Uniform	Ad-hoc
Default	1322	3648	2283	1296
EquivEdges	1203	1687	805	769
FolNonTerm	1175	3118	1159	951
LoopLimit	1322	3646	2283	1296
EquivEdges FolNonTerm	1119	1950	867	772
EquivEdges LoopLimit	1203	1612	805	768
FolNonTerm LoopLimit	1175	3116	1159	949
EquivEdges FolNonTerm LoopLimit	1118	1929	867	772

Table 9.2: Comparison of unrepairable errors for the Bronnikov grammar.

for both grammars.

It is important to note that the two tables do not describe the complete picture regarding comparisons between different choices of grammar, cost assignment, and pruning method. For each table entry, there is an attempt to fix each of 59,643 errors. Only the small portion of errors which weren't able to be repaired are represented in the tables. Despite this fact, the numbers in the tables give a good estimate of the effectiveness of the algorithm; if one entry in the table has a lower number than another, then the assertion can be made with reasonable confidence that the method with the lower number is the better method.

Differences between two numbers of the table tend to underestimate the relative performance of each method. For example, Figure 9.1 compares the uniform cost assignment and the ad-hoc cost assignment for the *EquivEdges* algorithm using the Bronnikov grammar. Points below the diagonal line indicate a shorter repair using ad-hoc costs, above the diagonal line indicate a shorter repair for uniform costs. The figure corresponds to the two entries (805 and 769) in the *EquivEdges* row in Table 9.2. At the extremes of the figure are errors that were not repaired before the limit of 1,000,000 configurations, 805 points for uniform costs, and 769 points for *EquivEdges*.

While there is a small difference in the number of unrepairable errors (about 5%), the effective difference is much greater. There are a number of repairs that take a small number of configurations to repair using ad-hoc costs that take a large number of configurations using uniform costs. On the other hand, a few repairs perform worse using ad-hoc costs, showing that a trade-off exists with cost variations. Care must be taken to ensure that changing the cost assignment leads to more improvements in the number of configurations required to find a repairs, rather than fewer.

Figure 9.2 represents the same data as Figure 9.1 on a log-log scale, emphasising the easier-to-repair errors. It shows many of the shorter repairs are repaired faster using ad-hoc costs than with using uniform costs.

Although much has been made for the case of insert/delete costs when searching for a repair, it was found that uniform insert/delete costs were often better than the two other methodically derived cost assignments. Cost

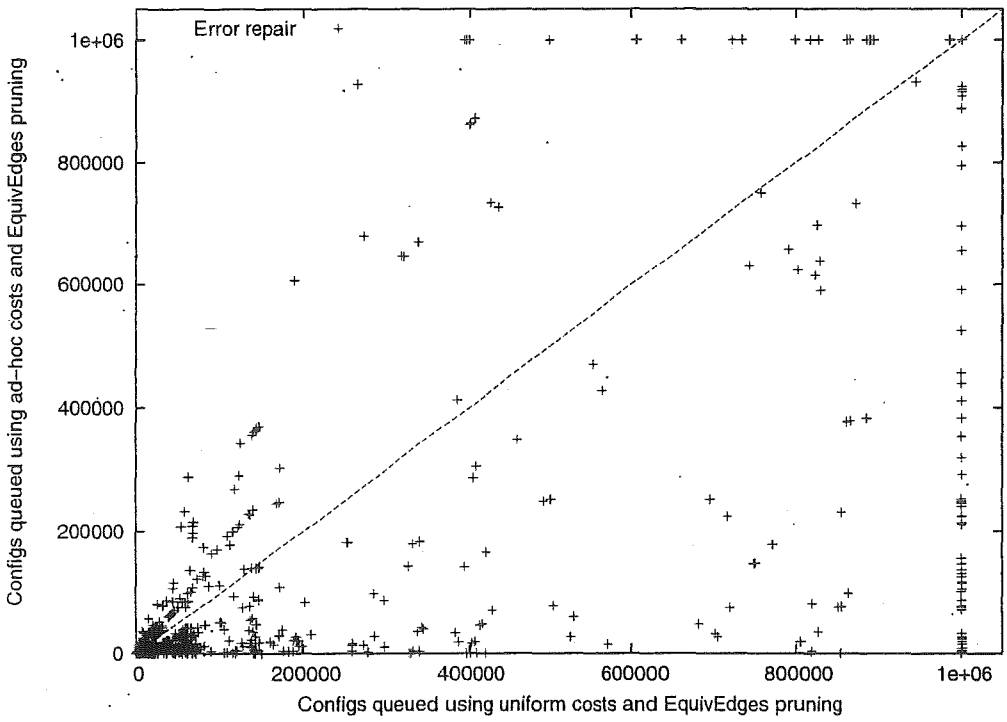

Figure 9.1: Comparison of ~~u~~niform costs and ad-hoc costs using the *EquivEdges* algorithm and the Bronnikov grammar.

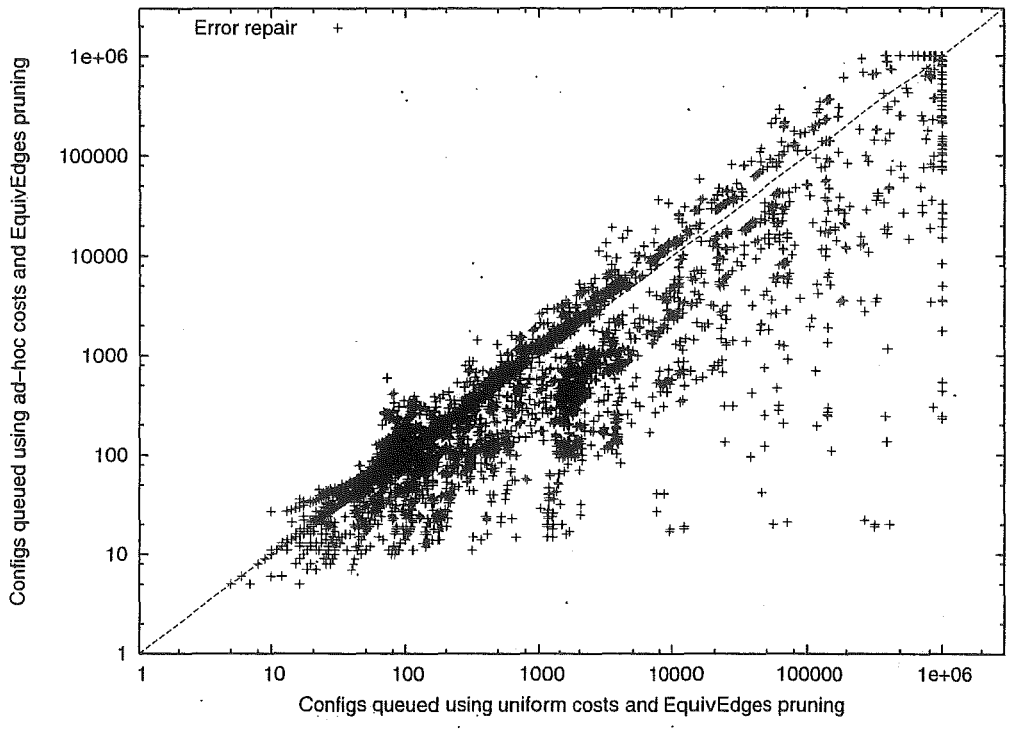

Figure 9.2: Comparison of uniform costs and ad-hoc costs using the *EquivEdges* algorithm and the Bronnikov grammar.

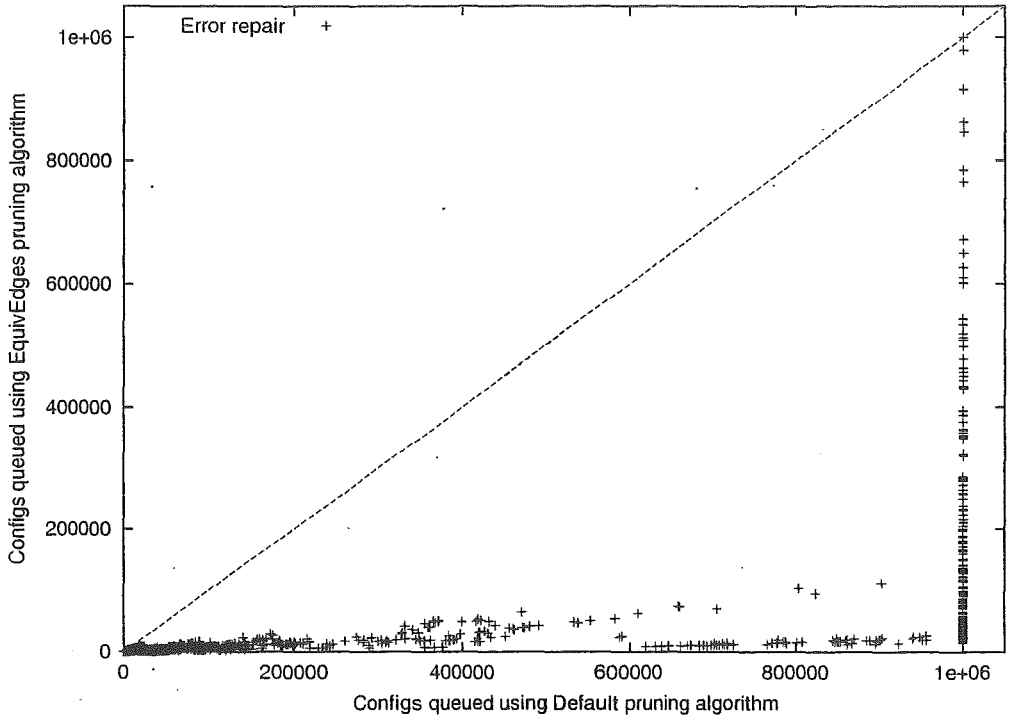

Figure 9.3: Comparison of *Default* pruning algorithm and *EquivEdges* pruning algorithm using ad-hoc costs and the Bäckerud grammar.

assignments contribute a large amount to the repairability of an error, as evidenced by the large difference between the numbers in Table 9.1. Often length-based costs in this table had more than ten times as many errors that were not able to be repaired as the ad-hoc method.

9.2 Comparison of the pruning algorithms

All relevant comparisons of the pruning algorithms were performed, but only the interesting comparisons are presented here. Each pruning algorithm is considered separately, and compared with the *Default* algorithm (no pruning). Finally, the performance of pruning algorithm combinations are compared.

9.2.1 *The EquivEdges algorithm*

The *EquivEdges* algorithm (Chapter 6) proved to be the most effective individually, in comparison with the *Default* (McKenzie) search. Figure 9.3 shows the improvement over the *Default* algorithm for the 59,643 repairs for the Bäckerud grammar. The large number of points near the bottom of the graph show those repairs which took a large number of configurations to repair using the *Default* algorithm yet were repaired very quickly using the *EquivEdges* pruning algorithm. Many of the repairs using the *EquivEdges* algorithm required fewer than 1/10 the number of configurations than with the *Default* algorithm.

There was only one case where another pruning algorithm outperformed the *EquivEdges* algorithm— frequency-based costs using the Bronnikov grammar had marginally fewer unrepairable errors using the *FolNonTerm* algorithm. Figure 9.4 shows the comparison of the two pruning algorithms for the Bronnikov grammar using frequency-based costs. The two algorithms perform nearly the same, with a slight advantage to *FolNonTerm*, consistent with the entries in Table 9.2 showing a few more unrepairable errors for the *EquivEdges* algorithm in this case.

9.2.2 *The FolNonTerm algorithm*

The *FolNonTerm* algorithm (Chapter 4) was effective at reducing the time taken to find repairs, except for length-based costs using the Bäckerud grammar (see Table 9.1), where performance was worse than when no pruning at all was used. The *FolNonTerm* had the largest beneficial effect with the ad-hoc costs applied to the Bäckerud grammar, and with uniform costs applied to the Bronnikov grammar.

Because this pruning algorithm follows non-terminals in the parser in order to prune future reductions, it relies on the fact that more reductions will be pruned than non-terminals followed. For shorter repairs, and poorly chosen costs, it is often the case that fewer reductions are performed than non-terminals followed, leading to longer repairs than the default algorithm.

Figure 9.5 shows the detail of the comparison between *FolNonTerm* and the default algorithm using ad-hoc costs and the Bronnikov grammar. The

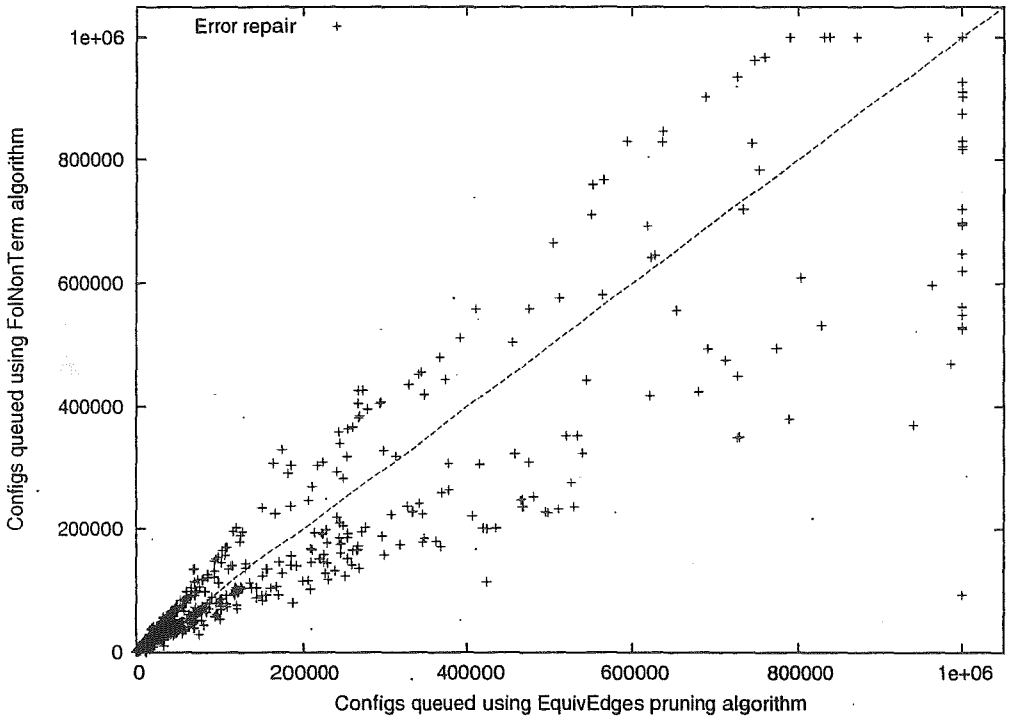

Figure 9.4: Comparison of *FolNonTerm* algorithm and *EquivEdges* algorithm, using frequency-based costs and Bronnikov grammar.

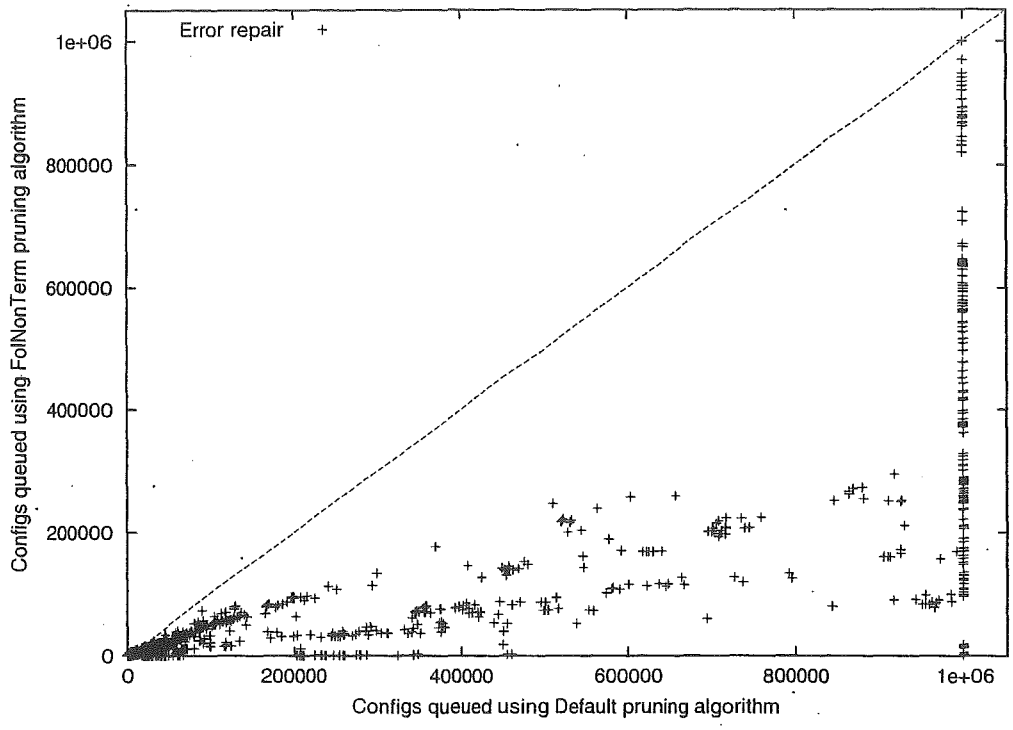

Figure 9.5: Comparison of *Default* algorithm and *FolNonTerm* algorithm, using ad-hoc costs and Bronnikov grammar.

FolNonTerm algorithm is clearly pruning a large amount of the search space in this comparison.

The *FolNonTerm* algorithm prunes less effectively than the *EquivEdges* algorithm as the comparison in Figure 9.6 shows.

9.2.3 The LoopLimit algorithm

The *LoopLimit* algorithm (Chapter 5) performed poorly. It showed almost no improvement on the *Default* algorithm using the Bäckerud grammar, and a very minor effect using the Bronnikov grammar. In almost all cases, *LoopLimit* was more effective when combined with another pruning algorithm. There was only one case where *LoopLimit* on its own made any difference to the number of unrepairable errors— length-based costs with the Bronnikov grammar. The detailed comparison is shown in Figure 9.7.

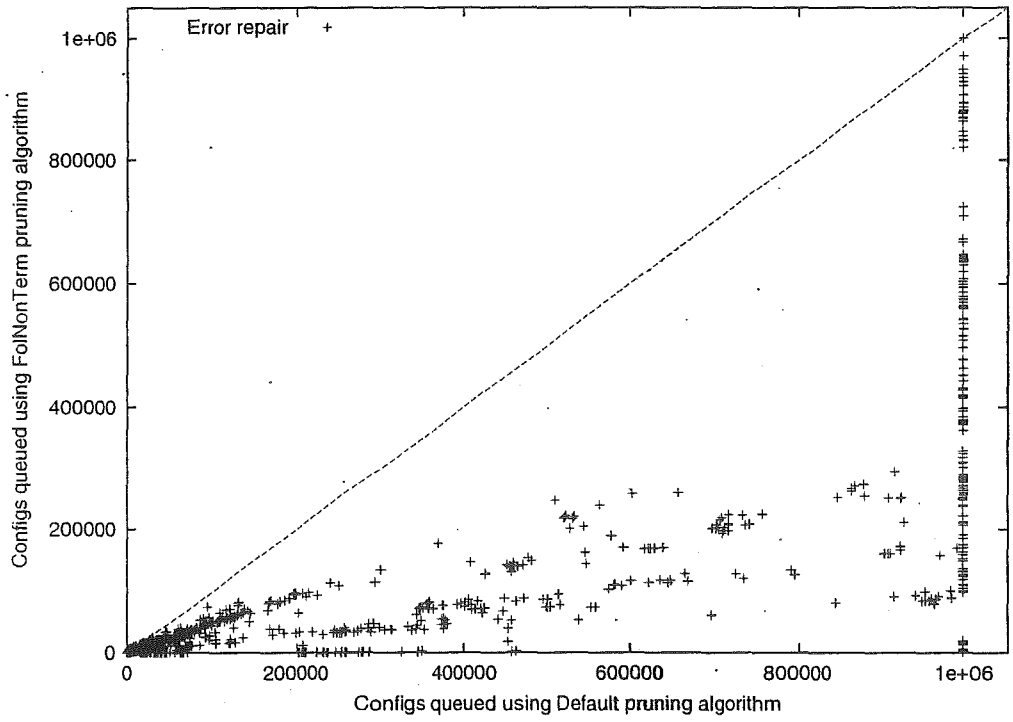

Figure 9.6: Comparison of *FolNonTerm* algorithm and *EquivEdges* algorithm, using ad-hoc costs and Bronnikov grammar.

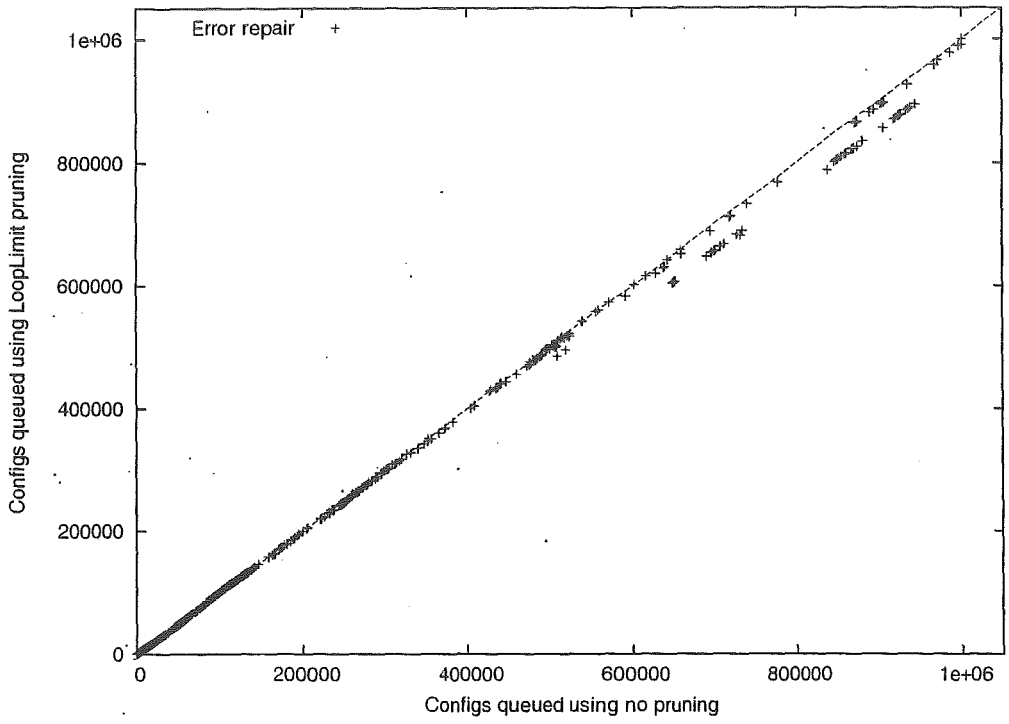

Figure 9.7: Comparison of *Default* algorithm and *LoopLimit* algorithm, using length-based costs and Bronnikov grammar.

The reason for the poor performance of *LoopLimit* is likely to be because, for a validation length v, limiting of loops only comes into effect after a loop has been traversed v times. Even for a validation length of three, the pruning opportunities for this algorithm are few before the configurations-queued limit of 1,000,000 is reached.

9.2.4 Algorithm combinations

There are four possible combinations of more than one algorithm (see Tables 9.1 and 9.2). Three of these combinations involve the *LoopLimit* algorithm, which has little effect on the number of configurations required to find a repair. Longer repairs, those requiring more than 100,00 configurations, where the pruning of the *LoopLimit* algorithm is greatest, showed a reduction in the number of configurations of only 0.04-5% for the Bäckerud

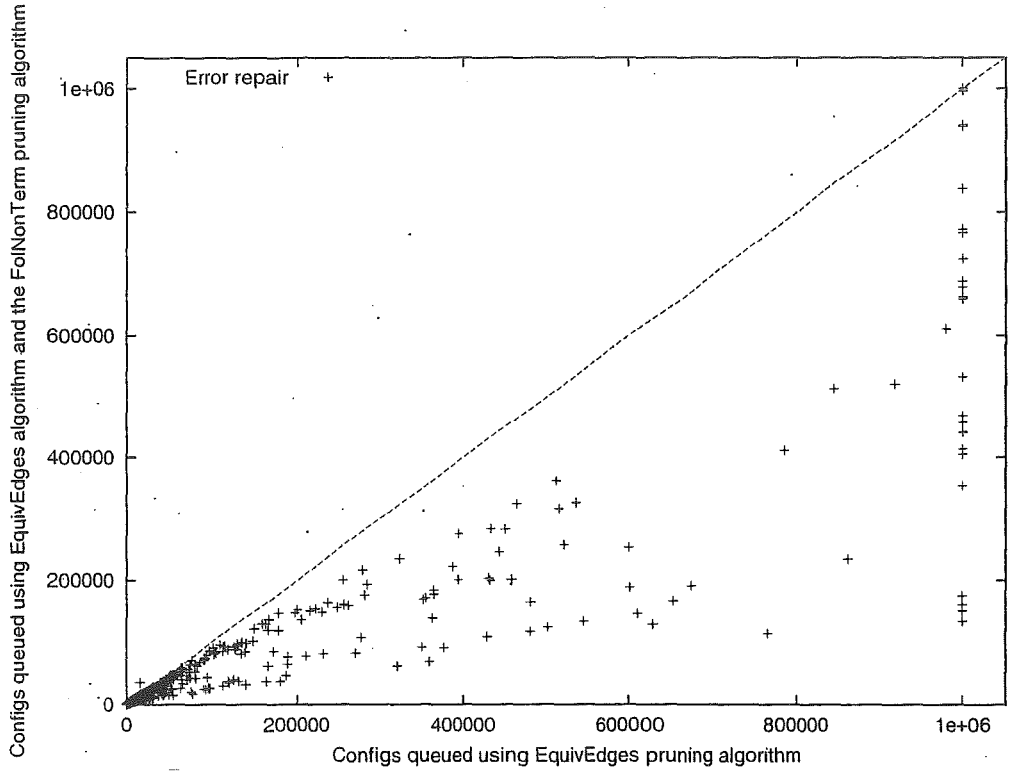

Figure 9.8: Comparison of *EquivEdges* algorithm and the combination of the *EquivEdges* and *FolNonTerm* algorithms, using ad-hoc costs and Bäckerud grammar.

grammar, and 0.1-7% reduction for the Bronnikov grammar.

As a consequence of the poor pruning ability of the *LoopLimit* algorithm, only the combination that does not include it, (the *EquivEdges*/*FolNonTerm* combination) is considered here.

The *EquivEdges*/*FolNonTerm* combination gives the best result for the Bäckerud grammar. The unrepairable errors are down to 85 from 131 for the *EquivEdges* algorithm with ad-hoc costs. Figure 9.8 compares the two algorithms across all repairs and shows a significant reduction in the number of configurations queued for the combination of both algorithms than for the *EquivEdges* algorithm alone. A few very easy repairs, however, take longer to repair with the addition of the *FolNonTerm* algorithm.

The result for the Bronnikov grammar contrast with the Bäckerud grammar in that the results for the combination of the two algorithms is worse than

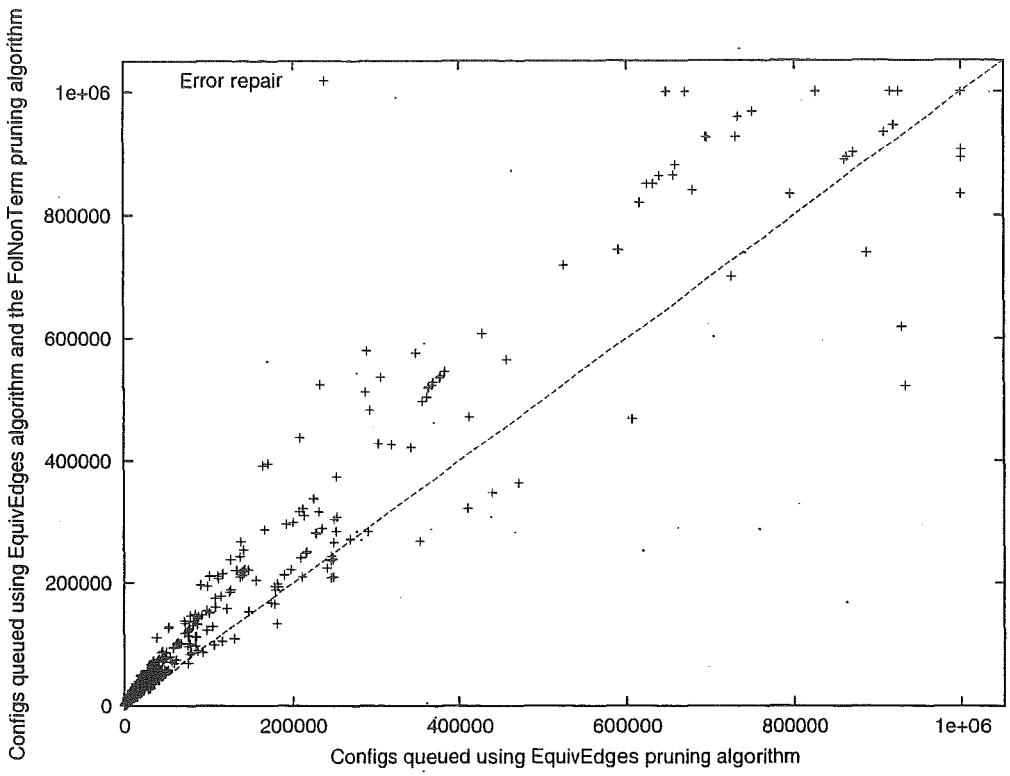

Figure 9.9: Comparison of *EquivEdges* algorithm and the combination of the *EquivEdges* and *FolNonTerm* algorithms, using ad-hoc costs and Bronnikov grammar.

for *EquivEdges* alone, with the exception of frequency-based costs. Figure 9.9 shows the comparison for the Bronnikov grammar showing that, while some errors are repaired faster, most are repaired slower. The reason for the difference may lie in the way the two grammars are structured (Section 8.5). The Bronnikov grammar has more non-terminals than the Bäckerud grammar, and the parser has a greater density of non-terminals in relation to terminals. The more hierarchical nature of the Bronnikov grammar leads to a parser with almost the same number of gotos as shifts (2295 and 2123 respectively). The flatter Bäckerud grammar produces a parser with 1272 gotos and 3588 shifts, almost three times as many shifts as gotos. The *FolNonTerm* algorithm, which follows each non-terminal, is therefore likely to queue more configurations in the Bronnikov grammar than in the Bäckerud grammar.

9.3 Comparison between the two grammars

From the two results tables, the Bäckerud grammar showed the most variation in its ability to repair errors, both across algorithms and cost assignments. For both frequency-based costs and length-based costs, the Bäckerud grammar was more extreme— it had worse worst values and better best values. For uniform costs, it performed worse than the Bronnikov grammar in all cases, yet for ad-hoc costs, it performed better in all cases. It is not clear why the differences in the grammars (the Bronnikov grammar uses no ϵ productions, and is more hierarchical than the Bäckerud grammar— see Section 8.5) would produce these effects.

Even though the ad-hoc costs were derived from an analysis of the most difficult to repair programs from the Bäckerud grammar is not sufficient to explain the advantage that that grammar appears to have over the Bronnikov grammar. Of the 880 difficult to repair programs (using uniform costs and *EquivEdges* and *FolNonTerm* pruning algorithms) from the Bäckerud grammar that were examined to determine the ad-hoc costs (Section 8.6.4), 844 (96%) were also present in the corresponding table entry (with 867 unrepairable errors) for the Bronnikov grammar.

Because the grammars recognise two slightly different supersets of the Java language, occasionally a repair that was completed quickly in one grammar takes much longer than the other grammar, or is even not able to be repaired within the configuration queue limit.

Figure 9.10 shows a number of points at $y = 1e+06$ where one grammar finds a repair quickly, while the other grammar does not find a repair at all. These differences can be very large, sometimes by a factor of 10,000 or more. The cause of these large differences in the number of configurations queued in a repair attempt is the slightly different languages that each grammar accepts. What may be a valid repair (that is, parsing may continue for validation-length number of symbols) for one grammar may not be a valid repair for the other grammar. The percentage of errors where this was obviously the problem was lower than 1%.

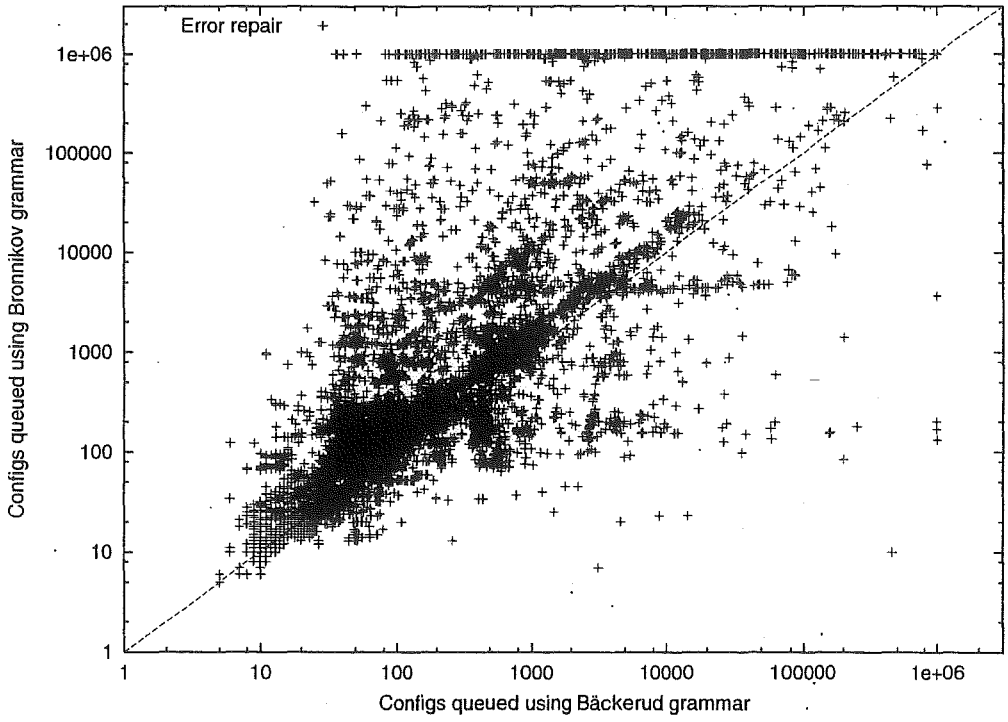

Figure 9.10: Loglog comparison of the configurations queued for the Bäckerud and Bronnikov grammars. Ad-hoc costs used, and *EquivEdges* and *FolNonTerm* algorithms enabled.

9.3.1 *Effects of grammar strictness on repair*

In general, the percentage of errors repaired may be a poor indicator of how good a grammar is for repairing errors. At one extreme, there could be a syntactically loose grammar where repairs are easy to find because the language defined by the grammar is a large superset of the actual language required. At the other extreme would be a grammar that very tightly defines the syntax of the language, making errors more difficult to repair, but giving better quality repairs. The measurement of 'quality', while extremely important (what good, after all, is a successful repair that is not useful for providing an accurate error report) is very subjective and difficult to measure [63].

9.4 *Results summary*

The results show that choice of insert/delete costs has a large effect on the time taken to find a repair. A method similar to the ad-hoc method described in Section 8.6.4 is likely to lead to faster repairs than costs derived by an automatic method. No variation in costs (uniform costs) can quite possibly lead to faster repairs than costs derived by an automatic method.

If only one pruning algorithm can be implemented by a compiler writer, it should the *EquivEdges* pruning algorithm, although it is somewhat complicated. The *FolNonTerm* pruning algorithm is much easier to implement than *EquivEdges*, but does not prune as effectively. The two algorithms in combination are best. The *LoopLimit* algorithm is not effective enough in pruning the search space to be recommended, though may prove useful in unusual grammars[3] with many interacting loops.

Finally, it does not appear that any special consideration of error recovery should be taken into account when constructing a grammar, as differences in grammar structure tend to have little impact on the ability to find a repair.

[3] Such a grammar would be unlikely to resemble a typical programming language.

Chapter X

Conclusion

This thesis presents three pruning algorithms, *FolNonTerm*, *LoopLimit*, and *EquivEdges*, for reducing the search space while performing a locally least-cost error repair. The three pruning algorithms are independent of each other; they may be used in any combination when searching for a repair. Experiments conducted with a large number (59,643) of incorrect Java programs collected from novice programmers show a large reduction in the number of configurations required to find a repair when using the *FolNonTerm* and the *EquivEdges* pruning algorithms in combination. In contrast, the *LoopLimit* algorithm performed poorly, and would not be worth the trouble of implementing.

A more efficient priority queue implementation specific to the needs of maintaining configurations in a locally least-cost search is presented with linear time complexity for insert and remove-min operations, improving on the $O(\log n)$ time remove-min operation for a generalised tree-based priority queue implementation.

Two grammars for Java were examined, each using a different style for representing syntactic constructs. One grammar was more hierarchical and had no ϵ productions, the other was more flat, with many ϵ productions. There were no major differences between the grammars when searching for a locally least-cost repair.

The assignment of costs to insert and delete operations was discovered to have a significant effect on the ability of a locally least-cost algorithm to find a repair. In particular, automatic methods that have been suggested in the literature, with costs based on the length or frequency of tokens, were found to be worse than uniform costs. An ad-hoc cost arrangement based on

analysis of difficult to repair programs was presented and gives significantly better performance than uniform costs.

The best combination for locally least-cost error repair in LR parsers is to use both *FolNonTerm* and the *EquivEdges* pruning algorithms to reduce the search space, along with the ad-hoc method of assigning costs described in this thesis.

The future work section (Section 11) suggests a more efficient searching method which would also automatically incorporate the three pruning methods described.

Chapter XI

Future work

11.1 A new, more efficient repair algorithm

Towards the end of the Ph.D. research, an idea for a new error repair algorithm surfaced that combined all the benefits of the three pruning algorithms introduced in this thesis, and is also able to repair errors faster, with the possible drawback that finding the least-cost error may be more difficult in some cases. Unfortunately, there was no time to implement it.

When an error occurs, the new algorithm, *shortest forward path*, first determines the set of stacks the parser must have to resume parsing. It then looks up a pre-calculated table of shortest paths to help transform the parse stack (by inserting symbols) to a stack from which parsing may resume.

The static part of the algorithm (computed at parser-generation time) consists of a table of shortest forward paths from each state in the generated parser to all other states that are reachable via a sequence of shifts and gotos, and also a list of possible reductions back past the state into the stack. A shortest forward path table for a Java grammar can be compacted to under 150Kb, small enough to be easily added to modern compilers.

For the dynamic part of the algorithm (when an error occurs during a parse), the set of stack suffixes of the parser are found from which the next v symbols are able to be parsed. v is the validation length, typically about 3. This can be done reasonably efficiently; the related problem of substring recognition of LR(k) languages can be done in linear time [12]. If there exist no stack suffixes which permit the next v symbols to be parsed, the next input symbol (the first of the v symbols) is deleted, and the algorithm tries again. Note that this could also be statically encoded in a table of all possible v-length substrings of the language at parser generation time. Such a table

is likely to be excessively large for all but the smallest grammars.

Once a set of stack suffixes is found that are able to successfully resume parsing the next few symbols, the problem is reduced to finding the shortest (or least cost) path that modifies the current stack into a stack where the suffix matches one of the suffixes which allow parsing to resume.

The table of shortest forward paths is consulted to see if there is a forward path (shifts and gotos only; no reductions) from the top of the existing stack to the bottom-most state of any of the resumable stack suffixes. If there is a path, then it represents a valid repair, although not necessarily a least-cost repair. If no forward path exists, or more searching is necessary to find the least cost repair, the list of reductions back into the stack are performed, resulting in a set of possible stacks. Each of the stacks in the set is checked to see if there is a valid path from the state at the top of the stack to a resumable stack. The process continues for as long as necessary.

This algorithm supersedes the three pruning algorithms developed as part of this thesis:

- *FolNonTerm*: Non terminals are followed automatically. No special consideration is necessary.

- *LoopLimit*: Any loops that must be traversed for a repair are found in the substring recognition phase: For an expression, with a rule `E → ( E )`, successfully resuming the input '))' may require insertion of up to three '(' symbols. Each '(' is possibly on a looping edge. All stack suffixes that are found that can immediately parse '))' will necessarily include the sequence of states that parse '(('. That is, any loops that must be traversed for a repair will be present in the resumable stack suffix. If a path is found to the bottom of that stack suffix (from the table of shortest forward paths), then the symbols to build up the remainder of the stack suffix (including the loop) are directly known and can be inserted immediately. The table of shortest forward paths has no loops by definition.

- *EquivEdges*: Because there is ever only a maximum of one path from one state to another state in the table of shortest forward paths, know-

ing that two edges are equivalent becomes irrelevant. Even though there might be more than one shortest path between two states, only one is chosen for the table.

Another benefit of this algorithm is finding out in advance whether the next v symbols are a valid substring. No effort is wasted in checking whether a possible repair can parse the next v symbols if that substring cannot be parsed in any context.

Performance is likely to be very good. It is conjectured that all of the errors present in the collection of Java programs would be repaired in reasonable time using this algorithm.

A difficulty in this algorithm is termination. Knowing when the least-cost repair is found could prove problematic. It is possible that other non-least cost repairs would be found first. In particular, the algorithm as described favours insertion over deletion when the next v symbols are a legal substring. A cost assignment where deletes had a lower cost than inserts could prove problematic. Even if finding the least-cost repair proves difficult for a particular case, having an almost least-cost repair would be more useful than having no repair at all.

11.2 Replacement edit operation

Currently, only insertions and deletions are considered. It might be worthwhile to consider replacement of one symbol with another. For example, it might be useful to assign a cost to the replacement of '=' with '==' for a language such as Java (which, unlike C, does not consider assignment statements valid expressions). Such a cost one might reasonably assume to be less than the sum of the delete cost of '=' and the insert cost of '=='.

11.3 Other LR-related parsing methods

The algorithms presented in this thesis could also be extended to generalised LR parsing (also called Tomita [73] parsing). Bison is currently being extended by its authors to generate GLR parsers. An examination of the issues

particular to error repair for generalised LR parsing would be interesting. In particular, handling ambiguities may require some careful thought.

Also, applying error recovery in the context of Kannapin's recent work [47] on minimal LR parsers, where the size of the parsing automata is minimised, could lead to some interesting results. The smaller size may lead to a smaller search space, but less contextual information per state may complicate the search.

11.4 Improvements in the algorithm for finding equivalent edges

The algorithm presented in this thesis for finding equivalent edges, while finding many such edges (Figure 6.6), still has some limitations (Section 6.5): some cases of left-recursion , different non-terminals deriving the same set of strings, and certain patterns of reductions, can each cause some equivalent edges to not be found. It is not clear how many equivalent edges are missed because of these limitations, although in those parsers where 70-75% of edges were eliminated it is quite possible that few other equivalent edges remain.

It would be interesting to extend the algorithm to cope with these constructs.

11.5 Syntax error tracking and analysis

Tracking the syntax errors an individual makes over time would provide many insights into what areas of syntax novice programmers find most troublesome at different points in their learning. Such knowledge would be helpful to programming language instructors, language designers, and language-aware editors.

Also, it has been assumed that experienced programmers make errors which are more likely to be easily repaired. This makes sense, as an experienced programmer has a more complete mental model of the syntax of the programming language, and is less likely to make gross errors, such as those described in Section 8.6.4. It would be interesting to collect a large sample from experienced programmers and compare the errors with those from novice programmers.

11.6 Automatic error generation

Effective testing of error recovery methods requires a large collection of erroneous programs on which to test. Collecting real programs can take a long time (the collection of Java programs used in this thesis was collected over three years), and has privacy problems associated with real programs, but the errors contained therein are guaranteed to be genuine (discounting malicious compiler users).

Another method worth investigating is inserting artificial errors in programs generated automatically. This would only be useful if the randomly generated errors could approximate real errors. Generating strings at random from a context free grammar is certainly possible [58], but the question of how realistic artificial errors inserted into artificial programs can be is unanswered. If a method can be found that can give results shown to be reasonably close to real errors for the purposes of error recovery, then there would be no need for tedious collection of programs. Bizarre errors[1], though, are likely to remain the domain of humans.

[1] For example, FORTRAN program snippets within a Java method. See Section 8.6.4 for more.

Appendix A

Bäckerud Java grammar

This grammar was found on the internet at the URL:

`http//www.ifeb.se/gram.y`

Unfortunately, the grammar is no longer available on the Internet, and the web site `http//www.ifeb.se` no longer exists.

It is available from the `comp.compilers` usenet group archive ftp site:

`ftp://iecc.com/pub/file/java1.y`

```
%token ERROR   /* used by the lexer. */

%token ABSTRACT BOOLEAN BREAK BYTE CASE CATCH CHAR CLASS CONTINUE DEFAULT
%token DO DOUBLE EXTENDS FALSE_TOKEN FINAL FINALLY FLOAT FOR IF IMPLEMENTS
%token IMPORT INSTANCEOF INT INTERFACE LONG NATIVE NULL_TOKEN PACKAGE
%token PRIVATE PROTECTED PUBLIC RETURN SHORT STATIC SUPER SWITCH SYNCHRONIZED
%token THIS THROW THROWS TRANSIENT VOLATILE TRUE_TOKEN TRY VOID WHILE

%token INT_LITERAL CHARACTER_LITERAL
%token LONG_LITERAL
%token FLOAT_LITERAL
%token DOUBLE_LITERAL
%token SYMBOL STRING_LITERAL

%nonassoc NOT_AN_OPERATOR
%right SHIFT_RIGHT_EQUALS FILL_SHIFT_RIGHT_EQUALS SHIFT_LEFT_EQUALS \
       ADD_EQUALS SUB_EQUALS MUL_EQUALS DIV_EQUALS MOD_EQUALS AND_EQUALS \
       XOR_EQUALS OR_EQUALS '='
%right '?' ':'
%left OR
%left AND
%left '|'
%left '&'
%left '^'
%left EQUAL_COMPARE NOT_EQUAL
%left LTEQ GTEQ INSTANCEOF '<' '>'
%left BITSHIFT_RIGHT FILL_SHIFT_RIGHT SHIFT_LEFT
%left '+' '-'
```

```
%left '*' '/' '%'
%nonassoc CAST
%nonassoc INCR DECR '!' '~' UMINUS UPLUS
%nonassoc POST_INCR POST_DECR
%nonassoc NEW
%nonassoc '(' ')' '[' ']'
%left '.'

/* thse two are used to fix the standard dangling 'else' problem: */
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

/*
 * There should be three shift-reduce conflicts in the following code.  They
 * are essentially the same problem, and 'fixing' them would be extremely
 * painful, particularly given that bison's default behavior (shift) does the
 * right thing. In all three cases, expressions can end with a
 * 'qualifiedSymbol' which is defined to be   SYMBOL ('.' SYMBOL)*
 * and this conflicts with a situation where expression can itself be followed
 * by    '.' SYMBOL
 * By always shifting, we are sticking all of those extra .SYMBOL markers
 * onto the qualifiedSymbol, which is what we want to do anyway.
 * Fixing this would entail modifying every single expression to always
 * terminate the right way, which would be really difficult given that I
 * already had to do this for the beginning of expressions to disambiguate
 * statements from casts that start with a type.
 */
%expect 3

%%

compilationUnit : optPackage importList typeDeclarationList
    ;

optPackage :
    | PACKAGE qualifiedSymbol ';'
    ;

importList :
    | importList import
    ;

import : IMPORT qualifiedSymbol '.' '*' ';'
    | IMPORT qualifiedSymbol ';'
    ;

typeDeclarationList :
    | typeDeclarationList typeDeclaration
    ;
```

```
typeDeclaration : ';'
    | classDeclaration
    | interfaceDeclaration
    ;

classDeclaration :
    classModifierList CLASS simpleSymbol extends interfaces
    classBlock
    ;

interfaceDeclaration : classModifierList INTERFACE simpleSymbol
    interfaceExtends classBlock
    ;

classModifierList :
    | classModifierList FINAL
    | classModifierList PUBLIC
    | classModifierList ABSTRACT
    ;

extends :
    | EXTENDS qualifiedSymbol
    ;

interfaceExtends :
    | EXTENDS qualifiedSymbolList
    ;

interfaces :
    | IMPLEMENTS qualifiedSymbolList
    ;

classBlock : '{' '}'
    | '{' fieldList '}'
    ;

fieldList : field
    | fieldList field
    ;

field : ';'
    | methodDeclaration
    | constructorDeclaration
    | modifierList variableDeclaration
    | staticInitializer
    ;

staticInitializer : modifierList compoundStatement
    ;
```

```
methodDeclaration : modifierList type simpleSymbol '(' optParameterList ')'
        optArrayBounds optThrows methodBlock
    ;

constructorDeclaration : modifierList simpleSymbol
       '(' optParameterList ')' optThrows '{'
       optConstructorStatements '}'
    ;

optThrows :
    | THROWS qualifiedSymbolList
    ;

optConstructorStatements :
    | statement statementList
    | SUPER '(' optArgumentList ')' ';' statementList
    | THIS '(' optArgumentList ')' ';' statementList
    ;

methodBlock : ';'
    | compoundStatement
    ;

optParameterList :
    | parameterList
    ;

parameterList : parameter
    | parameterList ',' parameter
    ;

parameter : type simpleSymbol optArrayBounds
    ;

variableDeclaration : partialVariable ';'
    ;

/*
 * This rule is structured a little awkwardly because each rule needs to
 * know the base type when it is being evaluated, since in a situation like
 * this:     int x=0, y=x;
 * I can't evaluate "x=0, y=x" as a normal list and then wait to add the 'int'
 * type to these variables, since 'x' has already been used once.
 */
partialVariable : type simpleSymbol optArrayBounds optInitializer
    | partialVariable ',' simpleSymbol optArrayBounds optInitializer
    ;

optInitializer :
    | '=' initializer
```

```
    ;

initializer : expression
    | '{' optVariableInitializerList '}'
    ;

optVariableInitializerList :
    | variableInitializerList optComma
    ;

variableInitializerList : initializer
    | variableInitializerList ',' initializer
    ;

optComma :
    | ','
    ;

compoundStatement : '{' statementList '}'
    ;

statementList :
    | statementList statement
    ;

statement : ';'
    | compoundStatement
    | expressionStatement ';'
    | variableDeclaration
    | IF '(' expression ')' statement     %prec LOWER_THAN_ELSE
    | IF '(' expression ')' statement ELSE statement
    | WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | BREAK ';'
    | BREAK simpleSymbol ';'
    | CONTINUE ';'
    | CONTINUE simpleSymbol ';'
    | RETURN ';'
    | RETURN expression ';'
    | FOR '('
      forInit optExpression ';' forIncr ')' statement
    | THROW expression ';'
    | SYNCHRONIZED '(' expression ')' statement
    | simpleSymbol ':' statement    %prec NOT_AN_OPERATOR
    | TRY compoundStatement catchList optFinally
    | TRY compoundStatement finally
    | SWITCH '(' expression ')' compoundStatement
    | CASE expression ':' statement
    | DEFAULT ':' statement
    ;
```

```
optExpression :
    | expression
    ;

forInit : ';'
    | expressionStatements ';'
    | variableDeclaration
    ;

forIncr :
   | expressionStatements
   ;

expressionStatements : expressionStatement
    | expressionStatements ',' expressionStatement
    ;

optFinally :
    | finally
    ;

finally : FINALLY compoundStatement
    ;

catchList : catchItem
    | catchList catchItem
    ;

catchItem : CATCH '('
      qualifiedSymbol simpleSymbol
      ')' compoundStatement
    ;

expression : qualifiedSymbol
    | nonSymbolExpression
    ;

symbolArrayExpression : qualifiedSymbolWithLBracket expression ']'
    ;

nonSymbolExpression : expressionStatement
    | expression '?' expression ':' expression
    | expression OR expression
    | expression AND expression
    | expression '|' expression
    | expression '&' expression
    | expression '^' expression
    | expression EQUAL_COMPARE expression
    | expression NOT_EQUAL expression
```

```
    | expression LTEQ expression
    | expression GTEQ expression
    | expression '<' expression
    | expression '>' expression
    | expression BITSHIFT_RIGHT expression
    | expression FILL_SHIFT_RIGHT expression
    | expression SHIFT_LEFT expression
    | expression '+' expression
    | expression '-' expression
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | '-' expression  %prec UMINUS
    | '+' expression  %prec UPLUS
    | '!' expression
    | '~' expression
    | expression INSTANCEOF type
    | cast
    | nonSymbolComplexPrimary
    | SUPER
    | THIS
    | NULL_TOKEN
    ;

nonSymbolComplexPrimary : literal
    | '(' nonSymbolExpression ')'
    | '(' qualifiedSymbol ')'
    | nonSymbolComplexPrimary '[' expression ']'
    | methodCall '[' expression ']'
    | nonSymbolExpression '.' simpleSymbol
    | symbolArrayExpression
    ;

expressionStatement : expression '=' expression
    | expression SHIFT_RIGHT_EQUALS expression
    | expression FILL_SHIFT_RIGHT_EQUALS expression
    | expression SHIFT_LEFT_EQUALS expression
    | expression ADD_EQUALS expression
    | expression SUB_EQUALS expression
    | expression MUL_EQUALS expression
    | expression DIV_EQUALS expression
    | expression MOD_EQUALS expression
    | expression AND_EQUALS expression
    | expression XOR_EQUALS expression
    | expression OR_EQUALS expression
    | INCR expression
    | DECR expression
    | expression INCR  %prec POST_INCR
    | expression DECR  %prec POST_DECR
    | methodCall
```

```
    | newExpression
    ;

methodCall : qualifiedSymbol '(' optArgumentList ')'
    | nonSymbolExpression '.' simpleSymbol '(' optArgumentList ')'
    ;

cast : '(' simpleType optArrayBounds ')' expression  %prec CAST
    | '(' qualifiedSymbol ')' expression   %prec CAST
    | '(' qualifiedSymbolWithLBracket ']' optArrayBounds ')'
      expression   %prec CAST
    ;

newExpression : NEW simpleType '[' allocationBounds
    | NEW qualifiedSymbol '[' allocationBounds
    | NEW qualifiedSymbol
    | NEW qualifiedSymbol '(' optArgumentList ')'
    ;

allocationBounds : expression ']'
    | expression ']' '[' allocationBounds
    | expression ']' '[' ']' optArrayBounds
    ;

optArrayBounds :
    | optArrayBounds '[' ']'
    ;

literal : INT_LITERAL
    | STRING_LITERAL
    | CHARACTER_LITERAL
    | LONG_LITERAL
    | FLOAT_LITERAL
    | DOUBLE_LITERAL
    | TRUE_TOKEN
    | FALSE_TOKEN
    ;

optArgumentList :
    | argumentList
    ;

argumentList : expression
    | argumentList ',' expression
    ;

type : qualifiedSymbolWithLBracket ']' optArrayBounds
    | qualifiedSymbol
    | simpleType optArrayBounds
    ;
```

```
qualifiedSymbolWithLBracket : qualifiedSymbol '['
    ;

simpleType : BOOLEAN
    | BYTE
    | CHAR
    | SHORT
    | INT
    | FLOAT
    | LONG
    | DOUBLE
    | VOID
    ;

modifierList :
    | modifierList PRIVATE
    | modifierList PUBLIC
    | modifierList PROTECTED
    | modifierList STATIC
    | modifierList TRANSIENT
    | modifierList VOLATILE
    | modifierList FINAL
    | modifierList ABSTRACT
    | modifierList NATIVE
    | modifierList SYNCHRONIZED
    ;

qualifiedSymbol : simpleSymbol
    | qualifiedSymbol '.' simpleSymbol
    ;

qualifiedSymbolList : qualifiedSymbol
    | qualifiedSymbolList ',' qualifiedSymbol
    ;

simpleSymbol : SYMBOL
    ;
```

Appendix B

Bronnikov Java grammar

This grammar was found on the internet at the URL:

`http://home.attbi.com/~bronnikov/java.html`

It is widely available, and is also on the `comp.compilers` usenet group archive ftp site (along with a specification for a lexer):

`ftp://iecc.com/pub/file/java1.1-grammar.shar.gz`

The grammar was changed slightly so the tokens that both grammars use are the same. In particular, specifying array dimensions was changed from a single token recognising both delimiters ('`[ ]`'), to the two delimiter characters themselves.

```
%{
/*-------------------------------------------------------------------
 * Copyright (C)
 *   1996, 1997, 1998 Dmitri Bronnikov, All rights reserved.
 *
 * THIS GRAMMAR IS PROVIDED "AS IS" WITHOUT ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE, OR NON-INFRINGMENT.
 *
 * Bronikov@inreach.com
 *
 *-------------------------------------------------------------------
 *
 * VERSION 1.06 DATE 20 AUG 1998
 *
 *-------------------------------------------------------------------
 *
 * UPDATES
 *
 * 1.06 Correction of Java 1.1 syntax
```

```
* 1.05 Yet more Java 1.1
*      <qualified name>.<allocation expression>
* 1.04 More Java 1.1 features:
*      <class name>.this
*      <type name>.class
* 1.03 Added Java 1.1 features:
*      inner classes,
*      anonymous classes,
*      non-static initializer blocks,
*      array initialization by new operator
* 1.02 Corrected cast expression syntax
* 1.01 All shift/reduce conflicts, except dangling else, resolved
*
*----------------------------------------------------------------
*
* PARSING CONFLICTS RESOLVED
*
* Some Shift/Reduce conflicts have been resolved at the expense of
* the grammar defines a superset of the language. The following
* actions have to be performed to complete program syntax checking:
*
* 1) Check that modifiers applied to a class, interface, field,
*    or constructor are allowed in respectively a class, inteface,
*    field or constructor declaration. For example, a class
*    declaration should not allow other modifiers than abstract,
*    final and public.
*
* 2) For an expression statement, check it is either increment, or
*    decrement, or assignment expression.
*
* 3) Check that type expression in a cast operator indicates a type.
*    Some of the compilers that I have tested will allow simultaneous
*    use of identically named type and variable in the same scope
*    depending on context.
*
*
*    C. Cerecke: Point 4 has been changed back to '[' ']' so the number
*    of tokens is consistent with the Bckerud grammar.
* 4) Change lexical definition to change '[' optionally followed by
*    any number of white-space characters immediately followed by ']'
*    to OP_DIM token. I defined this token as [\[]{white_space}*[\]]
*    in the lexer.
*
*----------------------------------------------------------------
*
* UNRESOLVED SHIFT/REDUCE CONFLICTS
*
* Dangling else in if-then-else
*
*----------------------------------------------------------------
```

```
 */
%}

%token ERROR /* used by the lexer. Added by C Cerecke 12/12/00 */

%token ABSTRACT
%token BOOLEAN BREAK BYTE /* BYVALUE */
%token CASE /* CAST */ CATCH CHAR CLASS /* CONST */ CONTINUE
%token DEFAULT DO DOUBLE
%token ELSE EXTENDS
%token FINAL FINALLY FLOAT FOR /* FUTURE */
%token /* GENERIC */ /* GOTO */
%token IF IMPLEMENTS IMPORT /* INNER */ INSTANCEOF INT INTERFACE
%token LONG
%token NATIVE NEW NULL
%token /* OPERATOR */ /* OUTER */
%token PACKAGE PRIVATE PROTECTED PUBLIC
%token /* REST */ RETURN
%token SHORT STATIC SUPER SWITCH SYNCHRONIZED
%token THIS THROW THROWS TRANSIENT TRY
%token /* VAR */ VOID VOLATILE
%token WHILE
%token OP_INC OP_DEC
%token OP_SHL OP_SHR OP_SHRR
%token OP_GE OP_LE OP_EQ OP_NE
%token OP_LAND OP_LOR
/*%token OP_DIM      use '[' ']' instead - C Cerecke 13/12/00*/
%token ASS_MUL ASS_DIV ASS_MOD ASS_ADD ASS_SUB
%token ASS_SHL ASS_SHR ASS_SHRR ASS_AND ASS_XOR ASS_OR
%token IDENTIFIER LITERAL /* BOOLLIT */

/* tokens not used: REST GOTO GENERIC OPERATOR OUTER FUTURE BYVALUE */

%start CompilationUnit


%%

TypeSpecifier
        : TypeName
        | TypeName Dims
        ;

TypeName
        : PrimitiveType
        | QualifiedName
        ;

ClassNameList
        : QualifiedName
```

```
        | ClassNameList ',' QualifiedName
        ;

PrimitiveType
        : BOOLEAN
        | CHAR
        | BYTE
        | SHORT
        | INT
        | LONG
        | FLOAT
        | DOUBLE
        | VOID
        ;

SemiColons
        : ';'
        | SemiColons ';'
        ;

CompilationUnit
        : ProgramFile
        ;

ProgramFile
        : PackageStatement ImportStatements TypeDeclarations
        | PackageStatement ImportStatements
        | PackageStatement                 TypeDeclarations
        |                  ImportStatements TypeDeclarations
        | PackageStatement
        |                  ImportStatements
        |                                   TypeDeclarations
        ;

PackageStatement
        : PACKAGE QualifiedName SemiColons
        ;

TypeDeclarations
        : TypeDeclarationOptSemi
        | TypeDeclarations TypeDeclarationOptSemi
        ;

TypeDeclarationOptSemi
        : TypeDeclaration
        | TypeDeclaration SemiColons
        ;

ImportStatements
        : ImportStatement
```

```
        | ImportStatements ImportStatement
        ;

ImportStatement
        : IMPORT QualifiedName SemiColons
        | IMPORT QualifiedName '.' '*' SemiColons
        ;

QualifiedName
        : IDENTIFIER
        | QualifiedName '.' IDENTIFIER
        ;

TypeDeclaration
        : ClassHeader '{' FieldDeclarations '}'
        | ClassHeader '{' '}'
        ;

ClassHeader
        : Modifiers ClassWord IDENTIFIER Extends Interfaces
        | Modifiers ClassWord IDENTIFIER Extends
        | Modifiers ClassWord IDENTIFIER       Interfaces
        |           ClassWord IDENTIFIER Extends Interfaces
        | Modifiers ClassWord IDENTIFIER
        |           ClassWord IDENTIFIER Extends
        |           ClassWord IDENTIFIER       Interfaces
        |           ClassWord IDENTIFIER
        ;

Modifiers
        : Modifier
        | Modifiers Modifier
        ;

Modifier
        : ABSTRACT
        | FINAL
        | PUBLIC
        | PROTECTED
        | PRIVATE
        | STATIC
        | TRANSIENT
        | VOLATILE
        | NATIVE
        | SYNCHRONIZED
        ;

ClassWord
        : CLASS
        | INTERFACE
```

```
        ;

Interfaces
        : IMPLEMENTS ClassNameList
        ;

FieldDeclarations
        : FieldDeclarationOptSemi
        | FieldDeclarations FieldDeclarationOptSemi
        ;

FieldDeclarationOptSemi
        : FieldDeclaration
        | FieldDeclaration SemiColons
        ;

FieldDeclaration
        : FieldVariableDeclaration ';'
        | MethodDeclaration
        | ConstructorDeclaration
        | StaticInitializer
        | NonStaticInitializer
        | TypeDeclaration
        ;

FieldVariableDeclaration
        : Modifiers TypeSpecifier VariableDeclarators
        |           TypeSpecifier VariableDeclarators
        ;

VariableDeclarators
        : VariableDeclarator
        | VariableDeclarators ',' VariableDeclarator
        ;

VariableDeclarator
        : DeclaratorName
        | DeclaratorName '=' VariableInitializer
        ;

VariableInitializer
        : Expression
        | '{' '}'
        | '{' ArrayInitializers '}'
        ;

ArrayInitializers
        : VariableInitializer
        | ArrayInitializers ',' VariableInitializer
        | ArrayInitializers ','
```

```
        ;

MethodDeclaration
        : Modifiers TypeSpecifier MethodDeclarator Throws MethodBody
        | Modifiers TypeSpecifier MethodDeclarator        MethodBody
        |           TypeSpecifier MethodDeclarator Throws MethodBody
        |           TypeSpecifier MethodDeclarator        MethodBody
        ;

MethodDeclarator
        : DeclaratorName '(' ParameterList ')'
        | DeclaratorName '(' ')'
        | MethodDeclarator '[' ']'
        ;

ParameterList
        : Parameter
        | ParameterList ',' Parameter
        ;

Parameter
        : TypeSpecifier DeclaratorName
        | FINAL TypeSpecifier DeclaratorName
        ;

DeclaratorName
        : IDENTIFIER
        | DeclaratorName '[' ']'
        ;

Throws
        : THROWS ClassNameList
        ;

MethodBody
        : Block
        | ';'
        ;

ConstructorDeclaration
        : Modifiers ConstructorDeclarator Throws Block
        | Modifiers ConstructorDeclarator        Block
        |           ConstructorDeclarator Throws Block
        |           ConstructorDeclarator        Block
        ;

ConstructorDeclarator
        : IDENTIFIER '(' ParameterList ')'
        | IDENTIFIER '(' ')'
        ;
```

```
StaticInitializer
        : STATIC Block
        ;

NonStaticInitializer
        : Block
        ;

Extends
        : EXTENDS TypeName
        | Extends ',' TypeName
        ;

Block
        : '{' LocalVariableDeclarationsAndStatements '}'
        | '{' '}'
        ;

LocalVariableDeclarationsAndStatements
        : LocalVariableDeclarationOrStatement
        | LocalVariableDeclarationsAndStatements LocalVariableDeclarationOrStatement
        ;

LocalVariableDeclarationOrStatement
        : LocalVariableDeclarationStatement
        | Statement
        ;

LocalVariableDeclarationStatement
        : TypeSpecifier VariableDeclarators ';'
        | FINAL TypeSpecifier VariableDeclarators ';'
        ;

Statement
        : EmptyStatement
        | LabelStatement
        | ExpressionStatement ';'
        | SelectionStatement
        | IterationStatement
        | JumpStatement
        | GuardingStatement
        | Block
        ;

EmptyStatement
        : ';'
        ;

LabelStatement
```

```
        : IDENTIFIER ':'
        | CASE ConstantExpression ':'
        | DEFAULT ':'
        ;

ExpressionStatement
        : Expression
        ;

SelectionStatement
        : IF '(' Expression ')' Statement
        | IF '(' Expression ')' Statement ELSE Statement
        | SWITCH '(' Expression ')' Block
        ;

IterationStatement
        : WHILE '(' Expression ')' Statement
        | DO Statement WHILE '(' Expression ')' ';'
        | FOR '(' ForInit ForExpr ForIncr ')' Statement
        | FOR '(' ForInit ForExpr         ')' Statement
        ;

ForInit
        : ExpressionStatements ';'
        | LocalVariableDeclarationStatement
        | ';'
        ;

ForExpr
        : Expression ';'
        | ';'
        ;

ForIncr
        : ExpressionStatements
        ;

ExpressionStatements
        : ExpressionStatement
        | ExpressionStatements ',' ExpressionStatement
        ;

JumpStatement
        : BREAK IDENTIFIER ';'
        | BREAK            ';'
        | CONTINUE IDENTIFIER ';'
        | CONTINUE            ';'
        | RETURN Expression ';'
        | RETURN            ';'
        | THROW Expression ';'
```

```
        ;

GuardingStatement
        : SYNCHRONIZED '(' Expression ')' Statement
        | TRY Block Finally
        | TRY Block Catches
        | TRY Block Catches Finally
        ;

Catches
        : Catch
        | Catches Catch
        ;

Catch
        : CatchHeader Block
        ;

CatchHeader
        : CATCH '(' TypeSpecifier IDENTIFIER ')'
        | CATCH '(' TypeSpecifier ')'
        ;

Finally
        : FINALLY Block
        ;

PrimaryExpression
        : QualifiedName
        | NotJustName
        ;

NotJustName
        : SpecialName
        | NewAllocationExpression
        | ComplexPrimary
        ;

ComplexPrimary
        : '(' Expression ')'
        | ComplexPrimaryNoParenthesis
        ;

ComplexPrimaryNoParenthesis
        : LITERAL
    /*  | BOOLLIT  c.cerecke: No BOOLLIT token. LITERAL includes true/false */
        | ArrayAccess
        | FieldAccess
        | MethodCall
        ;
```

```
ArrayAccess
        : QualifiedName '[' Expression ']'
        | ComplexPrimary '[' Expression ']'
        ;

FieldAccess
        : NotJustName '.' IDENTIFIER
        | RealPostfixExpression '.' IDENTIFIER
        | QualifiedName '.' THIS
        | QualifiedName '.' CLASS
        | PrimitiveType '.' CLASS
        ;

MethodCall
        : MethodAccess '(' ArgumentList ')'
        | MethodAccess '(' ')'
        ;

MethodAccess
        : ComplexPrimaryNoParenthesis
        | SpecialName
        | QualifiedName
        ;

SpecialName
        : THIS
        | SUPER
        | NULL
        ;

ArgumentList
        : Expression
        | ArgumentList ',' Expression
        ;

NewAllocationExpression
        : PlainNewAllocationExpression
        | QualifiedName '.' PlainNewAllocationExpression
        ;

PlainNewAllocationExpression
        : ArrayAllocationExpression
        | ClassAllocationExpression
        | ArrayAllocationExpression '{' '}'
        | ClassAllocationExpression '{' '}'
        | ArrayAllocationExpression '{' ArrayInitializers '}'
        | ClassAllocationExpression '{' FieldDeclarations '}'
        ;
```

```
ClassAllocationExpression
        : NEW TypeName '(' ArgumentList ')'
        | NEW TypeName '('              ')'
        ;

ArrayAllocationExpression
        : NEW TypeName DimExprs Dims
        | NEW TypeName DimExprs
        | NEW TypeName Dims
        ;

DimExprs
        : DimExpr
        | DimExprs DimExpr
        ;

DimExpr
        : '[' Expression ']'
        ;

Dims
        : '[' ']'
        | Dims '[' ']'
        ;

PostfixExpression
        : PrimaryExpression
        | RealPostfixExpression
        ;

RealPostfixExpression
        : PostfixExpression OP_INC
        | PostfixExpression OP_DEC
        ;

UnaryExpression
        : OP_INC UnaryExpression
        | OP_DEC UnaryExpression
        | ArithmeticUnaryOperator CastExpression
        | LogicalUnaryExpression
        ;

LogicalUnaryExpression
        : PostfixExpression
        | LogicalUnaryOperator UnaryExpression
        ;

LogicalUnaryOperator
        : '~'
        | '!'
```

```
        ;

ArithmeticUnaryOperator
        : '+'
        | '-'
        ;

CastExpression
        : UnaryExpression
        | '(' PrimitiveTypeExpression ')' CastExpression
        | '(' ClassTypeExpression ')' CastExpression
        | '(' Expression ')' LogicalUnaryExpression
        ;

PrimitiveTypeExpression
        : PrimitiveType
        | PrimitiveType Dims
        ;

ClassTypeExpression
        : QualifiedName Dims
        ;

MultiplicativeExpression
        : CastExpression
        | MultiplicativeExpression '*' CastExpression
        | MultiplicativeExpression '/' CastExpression
        | MultiplicativeExpression '%' CastExpression
        ;

AdditiveExpression
        : MultiplicativeExpression
        | AdditiveExpression '+' MultiplicativeExpression
        | AdditiveExpression '-' MultiplicativeExpression
        ;

ShiftExpression
        : AdditiveExpression
        | ShiftExpression OP_SHL AdditiveExpression
        | ShiftExpression OP_SHR AdditiveExpression
        | ShiftExpression OP_SHRR AdditiveExpression
        ;

RelationalExpression
        : ShiftExpression
        | RelationalExpression '<' ShiftExpression
        | RelationalExpression '>' ShiftExpression
        | RelationalExpression OP_LE ShiftExpression
        | RelationalExpression OP_GE ShiftExpression
        | RelationalExpression INSTANCEOF TypeSpecifier
```

```
        ;

EqualityExpression
        : RelationalExpression
        | EqualityExpression OP_EQ RelationalExpression
        | EqualityExpression OP_NE RelationalExpression
        ;

AndExpression
        : EqualityExpression
        | AndExpression '&' EqualityExpression
        ;

ExclusiveOrExpression
        : AndExpression
        | ExclusiveOrExpression '^' AndExpression
        ;

InclusiveOrExpression
        : ExclusiveOrExpression
        | InclusiveOrExpression '|' ExclusiveOrExpression
        ;

ConditionalAndExpression
        : InclusiveOrExpression
        | ConditionalAndExpression OP_LAND InclusiveOrExpression
        ;

ConditionalOrExpression
        : ConditionalAndExpression
        | ConditionalOrExpression OP_LOR ConditionalAndExpression
        ;

ConditionalExpression
        : ConditionalOrExpression
        | ConditionalOrExpression '?' Expression ':' ConditionalExpression
        ;

AssignmentExpression
        : ConditionalExpression
        | UnaryExpression AssignmentOperator AssignmentExpression
        ;

AssignmentOperator
        : '='
        | ASS_MUL
        | ASS_DIV
        | ASS_MOD
        | ASS_ADD
        | ASS_SUB
```

```
        | ASS_SHL
        | ASS_SHR
        | ASS_SHRR
        | ASS_AND
        | ASS_XOR
        | ASS_OR
        ;

Expression
        : AssignmentExpression
        ;

ConstantExpression
        : ConditionalExpression
        ;
```

Appendix C

Java Token Frequencies

The first column shows the frequency of the tokens in the Java program collection (Section 8.2).

The second column is the number of bits required to represent the probability of the token appearing in a program based on the frequencies observed. The three tokens that did not occur at all were assigned an occurrence of one for the purposes of calculating the probability, to solve the zero-frequency problem [13].

The third column shows the percentage occurance of that token in the Java program collection, and the fourth column shows the token. The token `$undefined` (frequency of 33945) refers to characters of input (for example, the character `'@'` is not part of any token) the lexer could not tokenise.

Frequency	Cost	Percent	Token
21190884	2	29%	SYMBOL (Identifier)
6760532	3	9.2%	'('
6754312	3	9.2%	')'
6506179	3	8.9%	';'
4058243	4	5.5%	'.'
2769523	5	3.8%	'{'
2764510	5	3.8%	'}'
2690485	5	3.7%	'='
2583715	5	3.5%	INT_LITERAL
1695426	5	2.3%	PUBLIC
1524332	6	2.1%	INT

1208885	6	1.6%	'['
1208726	6	1.6%	']'
921530	6	1.3%	','
748869	7	1.0%	STRING_LITERAL
741015	7	1.0%	IF
694224	7	0.95%	RETURN
631533	7	0.86%	VOID
628674	7	0.86%	NEW
589791	7	0.80%	'+'
533259	7	0.73%	INCR
432948	7	0.59%	EQUAL_COMPARE
415616	7	0.57%	'-'
406539	7	0.55%	'<'
388694	8	0.53%	PRIVATE
350085	8	0.48%	FOR
306048	8	0.42%	CLASS
283616	8	0.39%	STATIC
258775	8	0.35%	ELSE
243333	8	0.33%	NULL_TOKEN
202120	9	0.28%	'*'
179309	9	0.24%	CHARACTER_LITERAL
159994	9	0.22%	BREAK
157637	9	0.21%	':'
151386	9	0.21%	'>'
131973	9	0.18%	CASE
129974	9	0.18%	BOOLEAN
128485	9	0.17%	TRUE_TOKEN
123205	9	0.17%	AND
117170	9	0.16%	CHAR
116054	9	0.16%	FALSE_TOKEN
115132	9	0.16%	FINAL

111205	9	0.15%	DOUBLE_LITERAL
101229	10	0.14%	WHILE
98862	10	0.13%	IMPORT
98347	10	0.13%	'/'
94808	10	0.13%	NOT_EQUAL
88996	10	0.12%	LTEQ
74951	10	0.10%	GTEQ
71124	10	0.097%	DECR
69434	10	0.095%	ABSTRACT
65503	10	0.089%	OR
62349	10	0.085%	DOUBLE
56051	10	0.076%	THROWS
55368	10	0.075%	EXTENDS
39906	11	0.054%	ADD_EQUALS
39516	11	0.054%	'!'
33945	11	0.046%	$undefined.
32910	11	0.045%	PROTECTED
31936	11	0.043%	'%'
29085	11	0.040%	CATCH
28467	11	0.039%	THIS
26883	11	0.037%	SWITCH
23920	12	0.033%	TRY
23346	12	0.032%	THROW
18229	12	0.025%	DEFAULT
12458	13	0.017%	SUPER
6644	13	<0.01%	IMPLEMENTS
5134	14	<0.01%	FLOAT
4565	14	<0.01%	'?'
2763	15	<0.01%	DIV_EQUALS
1644	15	<0.01%	CONTINUE
1438	16	<0.01%	SYNCHRONIZED

1380	16	<0.01%	MUL_EQUALS
1336	16	<0.01%	'&'
1052	16	<0.01%	DO
1008	16	<0.01%	'|'
754	17	<0.01%	MOD_EQUALS
617	17	<0.01%	INSTANCEOF
566	17	<0.01%	LONG
463	17	<0.01%	BYTE
394	18	<0.01%	SUB_EQUALS
394	18	<0.01%	FLOAT_LITERAL
202	18	<0.01%	INTERFACE
185	19	<0.01%	FINALLY
168	19	<0.01%	'^'
109	19	<0.01%	SHIFT_LEFT
77	20	<0.01%	FILL_SHIFT_RIGHT
69	20	<0.01%	FILL_SHIFT_RIGHT_EQUALS
54	20	<0.01%	OR_EQUALS
45	21	<0.01%	SHORT
34	21	<0.01%	BITSHIFT_RIGHT
23	22	<0.01%	LONG_LITERAL
21	22	<0.01%	PACKAGE
20	22	<0.01%	SHIFT_LEFT_EQUALS
20	22	<0.01%	AND_EQUALS
10	23	<0.01%	'~'
9	23	<0.01%	NATIVE
6	24	<0.01%	XOR_EQUALS
0	26	0%	VOLATILE
0	26	0%	TRANSIENT
0	26	0%	SHIFT_RIGHT_EQUALS

References

[1] AHO, A. V., AND JOHNSON, S. C. LR parsing. *ACM Computing Surveys 6*, 2 (1974), 99–124.

[2] AHO, A. V., AND PETERSON, T. G. A minimum-distance error-correcting parser for context-free languages. *SIAM Journal of Computing 1*, 4 (1972), 305–312.

[3] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1986.

[4] AHO, A. V., AND ULLMAN, J. D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.

[5] AJIT B, P., AND KIEBURTZ, R. B. Global context recovery: a new strategy for syntactic error recovery by table-driven parsers. *ACM Transactions on Programming Languages and Systems 2*, 1 (1980), 18–41.

[6] AMMANN, U. Error recovery in recursive descent prasers. In *State of the Art and Future Trends in Compilation* (INRIA, Paris, 1978), pp. 231–238.

[7] ANDERSON, S., AND BACKHOUSE, R. C. Locally least-cost error recovery in Earley's algorithm. *ACM Transactions on Programming Languages and Systems 3*, 3 (1981), 318–347.

[8] ANDERSON, S., AND BACKHOUSE, R. C. An alternative implementation of an insertion-only recovery technique. *Acta Informatica 18* (1982), 289–298.

[9] ANDERSON, S., BACKHOUSE, R. C., AND E.H. BUGGE, C. S. An assessment of locally least-cost error recovery. *The Computer Journal 26*, 1 (1983), 15–24.

[10] BACKHOUSE, R. C. *Syntax of Programming Languages.* Prentice-Hall, London, 1979.

[11] BARNARD, D. T., AND HOLT, R. C. Hierarchic syntax error repair for LR grammars. *International Journal of Comput. and Info. Sciences 11*, 4 (August 1982), 231–258.

[12] BATES, J., AND LAVIE, A. Recognizing substrings of LR(k) languages in linear time. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 1051–1077.

[13] BELL, T. C., CLEARY, J. G., AND WITTEN, I. H. *Text Compression.* Prentice Hall, 1990.

[14] BERTSCH, E., AND NEDERHOF, M.-J. A safe variant of the pruning technique in "error repair in shift-reduce parser". *ACM Transactions on Programming Languages and Systems 21*, 1 (1999), 1–10.

[15] BOULLIER, P. Automatic syntactic error recovery for LR parsers. In *State of the Art and Future Trends in Compilation* (Montpellier, France, 1978), vol. 2, pp. 239–252.

[16] BROWN, R. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31*, 10 (October 1988), 1220–1227.

[17] BURKE, M. G., AND FISHER, G. A. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems 9*, 2 (April 1987), 164–197.

[18] CHARLES, P. An LR(k) error diagnosis and recovery method. In *Second International Workshop on Parsing Technologies* (February 1991).

[19] CHARLES, P. *A practical method for constructing efficient LALR(k) parsers with automatic error recovery.* PhD thesis, Department of computer science, New York University, May 1991.

[20] CHYTIL, M. P., AND DEMNER, J. Panic mode without panic. In *14th International Colloquium on Automata, languages and programming* (1987), pp. 260–268.

[21] CLARE, G., HUTTON, B., AND THORNLEY, J. The design of automatic parsers with error recovery. Tech. rep., University of Auckland, 1987.

[22] CONWAY, R. W., AND WILCOX, T. R. Design and implementation of a diagnostic compiler for PL/I. *Communications of the ACM 16*, 3 (March 1973).

[23] CORBETT, R., AND STALLMAN, R. M. *Bison: Gnu parser generator.* Free Software Foundation, Cambridge, Mass., 1991.

[24] CORCHUELO, R., PÉREZ, J., RUIZ, A., AND TORO, M. Repairing syntax errors in LR parsers. *ACM Transactions on Programming Languages and Systems 24*, 6 (November 2002), 698–710.

[25] DAIN, J. Minimum distance error correction. Tech. rep., University of Warwick, June 1987.

[26] DAIN, J. A. *Automatic Error Recovery for LR Parsers in Theory and Practice.* PhD thesis, University of Warwick, September 1989.

[27] DEREMER, F. L. *Practical Translators for LR(k) Languages.* PhD thesis, Department of Electrical Engineering, MIT, 1969.

[28] DEREMER, F. L. Simple LR(k) grammars. *Communications of the ACM 14*, 7 (July 1971), 453–460.

[29] DION, B. A. *Locally least-cost error correctors for context-free and context-sensitive parsers.* PhD thesis, University of Wisconsin at Madison, 1982.

[30] EARLEY, J. An efficient context-free parsing algorithm. *Communications of the ACM 13*, 2 (1970), 94–102.

[31] ETSI. *The Testing and Test Control Notation version 3; Part1: TTCN-3 Core Language*, 2.2.1 ed., 2002.

[32] FISCHER, C. N., DION, B. A., , AND MAUNEY, J. A locally least-cost LR error-corrector. Tech. Rep. 363, University of Wisconsin-Madison, August 1979.

[33] FISCHER, C. N., AND MAUNEY, J. A simple, fast, and effective LL(1) error repair algorithm. *Acta Informatica* (1992), 109–120.

[34] FISCHER, C. N., MILTON, D. R., AND QUIRING, S. B. Efficient LL(1) error correction and recovery using only insertions. *Acta Informatica* (1980), 141–154.

[35] FISCHER, C. N., TAI, K. C., AND MILTON, D. R. Immediate error detection in strong LL(1) parsers. *Information Processing Letters 8*, 5 (1979), 261–266.

[36] FLOYD, R. Bounded context syntactic analysis. *Communications of the ACM 7*, 2 (February 1964), 62–67.

[37] FRANCE, J. E. L. *Syntax directed error recovery for compilers.* PhD thesis, Computer Science Department, University of Illinois, 1971.

[38] FYCOCK, S., AND LAZARUS, P. Syntax-directed correction of syntax errors. *Software—Practice and Experience 6*, 2 (1976), 207–219.

[39] GRAHAM, S. L., HALEY, C. B., AND JOY, W. N. Practical LR error recovery. *SIGPLAN Notices 14*, 8 (August 1979), 168–175.

[40] GRAHAM, S. L., AND RHODES, S. P. Practical syntactic error recovery. *Communications of the ACM 18*, 11 (1975), 639–650.

[41] GRUNE, D., AND JACOBS, C. *Parsing Techniques: A Practical Guide.* Ellis Horwood Ltd, August 1991.

[42] HAMMOND, K., AND RAYWARD-SMITH, V. J. A survey on syntactic error recovery and repair. *Computer Languages 9*, 1 (1984), 51–67.

[43] HARTMANN, A. C. *A Concurrent Pascal Compiler for Minicomputers.* Springer-Verlag, 1977.

[44] HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[45] JAMES, L. R. A syntax directed error recovery method. Master's thesis, Computer Systems Research Group, University of Toronto, Toronto, 1972.

[46] JOHNSON, S. C. YACC— yet another compiler compiler. Tech. rep., AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[47] KANNAPIN, S. *Reconstructing LR Theory to Eliminate Redundance, with an Application to the Construction of ELR Parsers.* PhD thesis, Department of Computer Sciences, Technical University of Berlin, 2001.

[48] KANTOROWITZ, E., AND LAOR, H. Automatic generation of useful syntax error messages. *Software—Practice and Experience 16*, 7 (1986), 627–640.

[49] KASAMI, T. An efficient recognition and syntax-analysis algorithm for context-free languages. Tech. rep., Air Force Cambridge Research Laboratory, Bedford, Mass., 1965.

[50] KIM, I., AND CHOE, K. Error repair with validation in LR-based parsing. *ACM Transactions on Programming Languages and Systems 23*, 4 (July 2001), 451–471.

[51] KINGSTON, J. *Algorithms and Data Structures.* Addison Wesley, 1990.

[52] KNUTH, D. E. On the translation of languages from left to right. *Information and Control 8*, 6 (December 1965), 607–639.

[53] LEINIUS, R. P. *Error derection and recovery for syntax directed systems.* PhD thesis, Computer Science Department, University of Wisconsin, Madison, 1970.

[54] LEVY, J. P. Automatic correction of syntax errors in programming languages. *Acta Informatica 4*, 3 (1975), 271–292.

[55] LEWIS, R. M., ROSENKRANTZ, D. J., AND STEARNS, R. E. *Compiler Design Theory.* Addison-Wesley, 1976.

[56] MAUNEY, J., AND FISCHER, C. ECP— an error correcting parser generator: User guide. Tech. Rep. 450, University of Wisconsin-Madison, October 1981.

[57] MAUNEY, J., AND FISCHER, C. N. A forward move algorithm for LL and LR parser. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, United States, 1982), ACM, pp. 79–87.

[58] MCKENZIE, B. J. Generating strings at random from a context free grammar. Tech. Rep. TR-COSC 10/97, Department of Computer Science, University of Canterbury, 1997.

[59] MCKENZIE, B. J., YEATMAN, C., AND DE VERE, L. Error repair in shift-reduce parsers. *ACM Transactions on Programming Languages and Systems 17*, 4 (July 1995), 672–689.

[60] MICKUNAS, M. D., AND MODRY, J. A. Automatic error recovery for LR parsers. *Communications of the ACM 21*, 6 (1978), 459–465.

[61] PEMBERTON, S. Comments on an error-recovery schame by hartmann. *Software—Practice and Experience 10*, 3 (1980), 231–240.

[62] PENNELLO, T. J., AND DEREMER, F. A forward move algorithm for LR error recovery. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages* (1978), pp. 241–254.

[63] RIPLEY, G. D., AND DRUSEIKIS, F. C. A statistical analysis of syntax errors. *Computer Languages 3* (1978), 227–240.

[64] SÉNIZERGUES, G. The equivalence problem for deterministic pushdown automata is decidable. In *Proceedings of the 24th International*

Conference on Automata, Languages and Programming. Lecture Notes in Computer Science (1997), Springer-Verlag, pp. 671–681.

[65] SOFTLAB. *Toolmaker reference manual.* Linköping, Sweden.

[66] SPENKE, M., MUHLENBEIN, H., MEVENKAMP, M., MATTERN, F., AND BEILKEN, C. A language independent error recovery method for LL(1) parsers. *Software—Practice and Experience 14*, 11 (1984), 1095–1107.

[67] SPERBER, M., AND THIEMANN, P. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems 22*, 2 (March 2000), 222–264.

[68] STIRLING, C. Follow set error recovery. *Software—Practice and Experience 15*, 3 (March 1985), 239–257.

[69] STIRLING, C. Decidability of DPDA equivalence. *Theoretical Computer Science*, 255 (2001), 1–31.

[70] SUDKAMP, T. A. *Languages and Machines: An Introduction to the Theory of Computer Science.* Addison-Wesley, 1996.

[71] TAI, K.-C. Syntactic error correction in programming languages. *IEEE transactions on software engineering 4*, 5 (1978), 414–423.

[72] TAI, K.-C. Locally minimum-distance correction of syntax errors in programming lnaguages. In *Proceedings of the ACM national conference* (New York, 1980), pp. 204–210.

[73] TOMITA, M. *Efficient Parsing for natural language: A fast algorithm for practical systems.* No. 0-89838-202-5. Kluwer Academic Publishers, 1986.

[74] TURNER, D. A. Error diagnosis and recovery in one pass compilers. *Information Process. Lett. 6*, 4 (1977), 113–115.

[75] UNGER, S. H. A global parser for context-free phrase structure grammars. *Communications of the ACM 11*, 4 (April 1968), 240–247.

[76] VILARES, M., DARRIBA, V., AND RIBADAS, F. Regional least-cost error repair. In *International Conference on Implementation and Application of Automata* (2000).

[77] WAGNER, R. A., AND FISCHER, M. J. The string-to-string correction problem. *Journal of the ACM 21*, 1 (1974), 168–173.

[78] WAGNER, R. A., AND FISCHER, M. J. The string-to-string correction problem. *Journal of the ACM 21*, 1 (1974), 168–173.

[79] WETHERELL, C., AND SHANNON, A. LR— automatic parser generator and LR(1) parser. *IEEE Transactions on Software Engineering 7*, 3 (1981), 274–278.

[80] WIRTH, N. Design of a pascal compiler. *Software—Practice and Experience 1* (1971), 309–333.

[81] WIRTH, N. *Algorithms + Data Structures = Programs.* Prentice Hall, 1976.

[82] YEATMAN, C. Error correction in LR parsers. Tech. rep., Department of Computer Science, University of Canterbury, 1992.

[83] YOUNGER, D. H. Recognition of context-free languages in time n^3. *Information Control 10*, 2 (1967), 189–208.